

U4 4. Übung

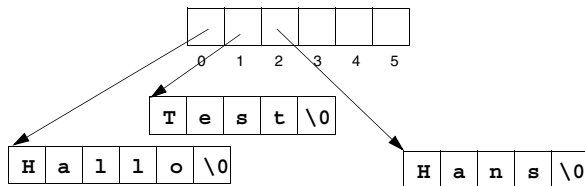
U4 4. Übung

- Besprechung Aufgabe 2
- dynamische Speicherallokation vs. Stackallokation
- Fehlerbehandlung
- Speicheraufbau eines Prozesses
- Prozesse: fork, exec, wait
- Aufgabe 4

2 wsort - Datenstrukturen (2. Möglichkeit)

U4-1 Aufgabe 2: Sortieren mittels qsort

- Array von Zeigern auf Zeichenketten



- Vorteile:
 - ◆ schnell, da nur Zeiger vertauscht werden (x86-32: 12 Byte pro Umordnung)
 - ◆ Zeichenketten können beliebig lang sein
 - ◆ sparsame Speichernutzung

U4-1 Aufgabe 2: Sortieren mittels qsort

U4-1 Aufgabe 2: Sortieren mittels qsort

1 wsort - Datenstrukturen (1. Möglichkeit)

- Array von Zeichenketten

H	a	l	l	o	\0	...	\0	T	e	s	t	\0	...	\0	H	a	n	s	...
0						...		100	101				...		201	202			

- Vorteile:
 - ◆ einfach
- Nachteile:
 - ◆ hoher Kopieraufwand (303 Bytes pro Umordnung)
 - ◆ Maximale Länge der Worte muss bekannt sein
 - ◆ Verschwendung von Speicherplatz

3 Speicherverwaltung

U4-1 Aufgabe 2: Sortieren mittels qsort

- Berechnung des Array-Speicherbedarfs
 - ◆ bei Lösung 1: Anzahl der Wörter * 101 * sizeof(char)
 - ◆ bei Lösung 2: Anzahl der Wörter * sizeof(char*)
- realloc:
 - ◆ Anzahl der zu lesenden Worte ist unbekannt
 - ◆ Array muss vergrößert werden: realloc
 - ◆ Bei Vergrößerung sollte man aus Effizienzgründen nicht nur Platz für ein neues Wort (Lösungsvariante 1) bzw. einen neuen Zeiger (Lösungsvariante 2) besorgen, sondern für mehrere.
 - ◆ Achtung: realloc kopiert möglicherweise das Array (teuer)
- Speicher sollte wieder freigegeben werden
 - ◆ bei Lösung 1: Array freigeben
 - ◆ bei Lösung 2: zuerst Wörter freigeben, dann Zeiger-Array freigeben

4 Vergleichsfunktion

- Problem: qsort erwartet folgenden Funktionszeigertyp:

```
int (*)(const void *, const void *)
```

- Lösung: "casten"

- ◆ innerhalb der Funktion, z.B. (Feld vom Typ char **):

```
int compare(const void *a, const void *b) {
    return strcmp(
        *(const char * const *) a,
        *(const char * const *) b
    );
}
```

- ◆ beim qsort-Aufruf:

```
int compare(const char * const *a, const char * const *b);
...
qsort(field, nel, sizeof(char *),
      (int (*)(const void *, const void *))compare);
```

U4-2 Verwendung dynamischer Speicherallokation

- Beispiel (aus Abgaben zu Aufgabe 2) mit dynamischer Heapallokation:

```
char *buffer = (char *) malloc(102 * sizeof(char));
if ( NULL == buffer ) {
    perror("malloc"); exit(EXIT_FAILURE);
}
while (fgets(buffer, 102, stdin) != NULL) {
    ... strcpy(somewhere_else, buffer); ...
}
free(buffer);
```

- teure Allokations- und Freigabeoperationen (siehe Aufgabe 3)
- erfordert Fehlerbehandlung
- viel Schreibarbeit
 - ◆ verschlechtert Code-Lesbarkeit
 - ◆ kostet Zeit wenn keine da ist (z.B. in der Klausur)

U4-2 Verwendung dynamischer Speicherallokation

- Alternative: (dynamische) Stackallokation

```
char buffer[102];

while (fgets(buffer, 102, stdin) != NULL) {
    ... strcpy(somewhere_else, buffer); ...
}
```

- Sehr effizient
 - ◆ Allokation: Stackpointer -= 102;
 - ◆ Freigabe: Stackpointer += 102;
- Keine Fehlerbehandlung durch das Programm
 - ◆ Stacküberlauf wird ggf. vom Betriebssystem erkannt (SIGSEGV)
- Implizite Freigabe beim Verlassen der Funktion
 - ◆ keine Speicherlecks möglich

U4-3 Fehlerbehandlung Reloaded

- Fehlermeldungen und Warnungen immer auf den Standardfehlerkanal
 - ◆ auch Warnungen des Programms: z.B. "Wort zu lang"
- Keine ungeforderten Ausgaben auf die Standardausgabe
 - ◆ wie z.B. "Hier kommt die sortierte Ausgabe"
 - ◆ beeinträchtigt die Verwendbarkeit des Programms in der Stapelverarbeitung
 - ◆ erschwert die Vergleichbarkeit mit anderen Lösungen

U4-3 Fehlerbehandlung Reloaded

- Signalisierung von Fehlern normalerweise durch Rückgabewert
- Nicht bei allen Funktionen möglich, z.B. **fgets(3)**

```
while (fgets(buffer, 102, stdin) != NULL) {
    ...
}

/* EOF oder Fehler? */
```

- Rückgabewert NULL sowohl im Fehlerfall als auch bei End-of-File

U4-3 Fehlerbehandlung Reloaded

- Erkennung im Fall von I/O-Streams mit **ferror(3)** und **feof(3)**

```
while (fgets(buffer, 102, stdin) != NULL) {
    ...

    /* EOF oder Fehler? */
    if (ferror(stdin)) {
        /* Fehler */
    }
}
```

U4-3 Fehlerbehandlung Reloaded

- Nicht in allen Fällen existieren solche Spezialfunktionen
- Allgemeiner Ansatz durch Setzen und Prüfen von **errno**

```
#include <errno.h>

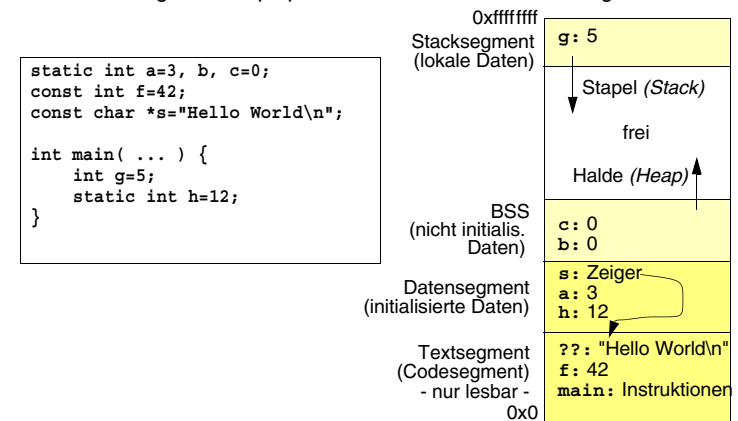
while (errno=0, readdir(...) != NULL) {
    ... /* keine break-Statements in der Schleife */
}

/* Ende oder Fehler? */
if (errno != 0) {
    /* Fehler */
    ...
}
```

- **errno=0** *unmittelbar* vor Aufruf der problematischen Funktion
→ **errno** wird nur im Fehlerfall gesetzt und bleibt sonst evtl. unverändert
- Abfrage der **errno** *unmittelbar* nach Rückgabe des pot. Fehlerwerts
→ **errno** könnte sonst durch andere Funktion verändert werden

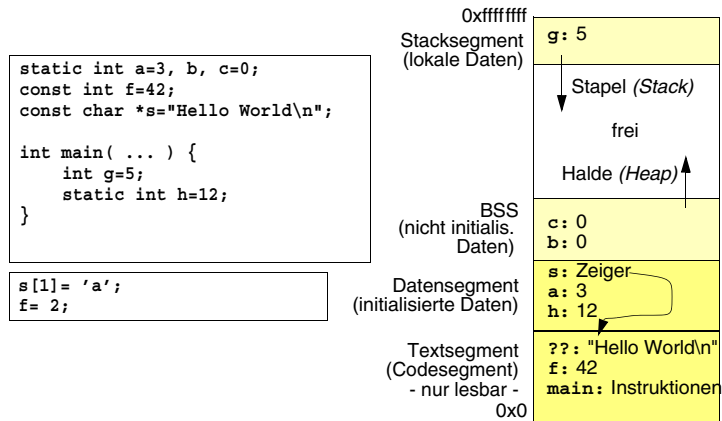
U4-4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente



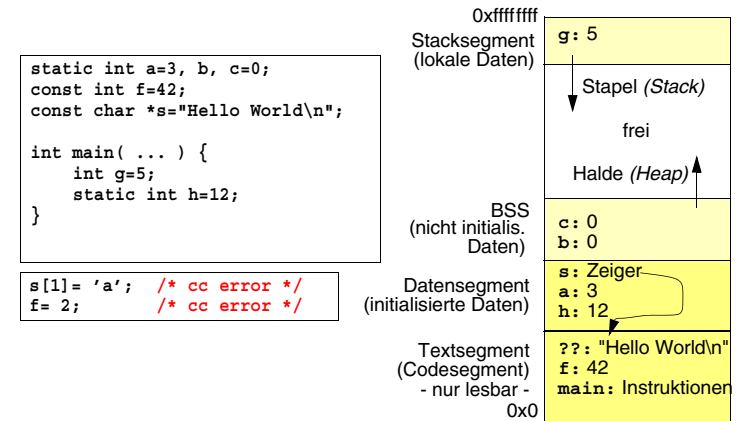
U4-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



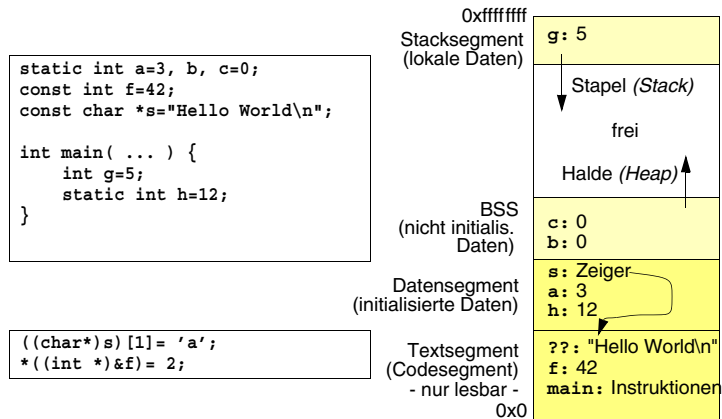
U4-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



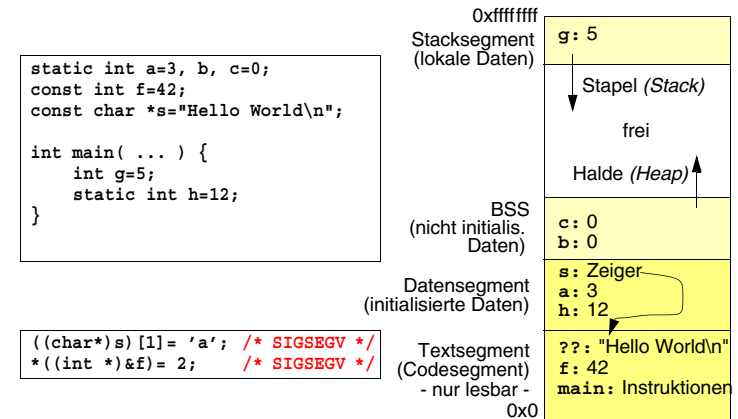
U4-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



U4-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

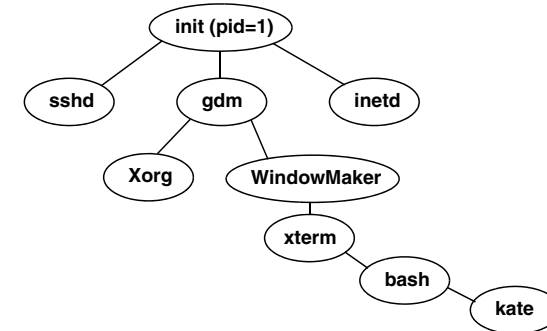


U4-5 Prozesse: Überblick

- Prozesse sind eine Ausführungsumgebung 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)

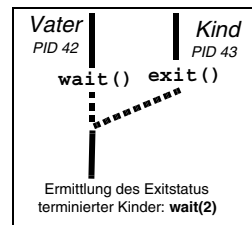
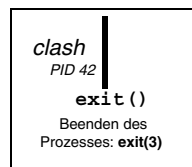
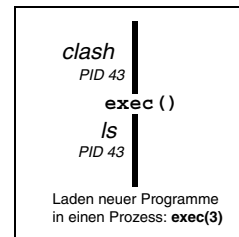
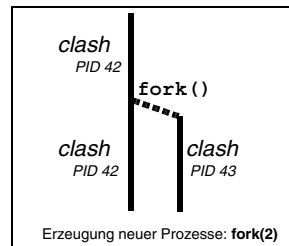
U4-5 UNIX-Prozesshierarchie

- 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**

U4-6 POSIX Prozess-Systemfunktionen

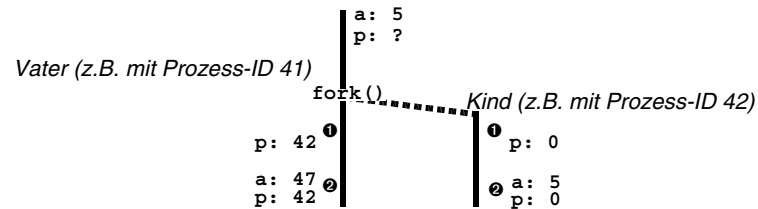


1 fork(2): Erzeugung eines neuen Prozesses

- Erzeugt einen neuen Kindprozess
- Exakte Kopie des Vaters...
 - ◆ Datensegment (neue Kopie, gleiche Daten)
 - ◆ Stacksegment (neue Kopie, gleiche Daten)
 - ◆ Textsegment (gemeinsam genutzt, da nur lesbar)
 - ◆ Filedescriptoren (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) 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 execlp(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 execlp(const char *file, char *const argv[]);
```

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

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)

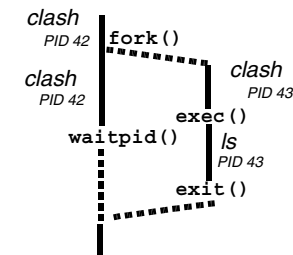
- Mächtigere Variante von `wait(2)`
`pid_t waitpid(pid_t pid, int *status, int options);`

- 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

U4-7 Aufgabe 4: 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 I-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

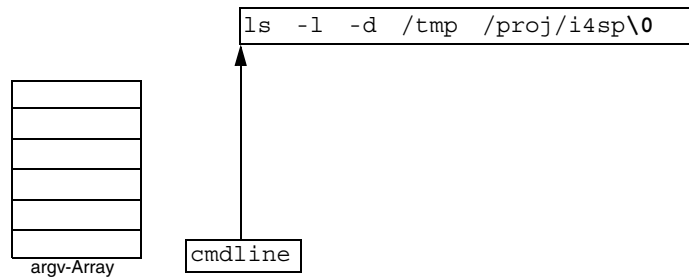
2 strtok

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

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

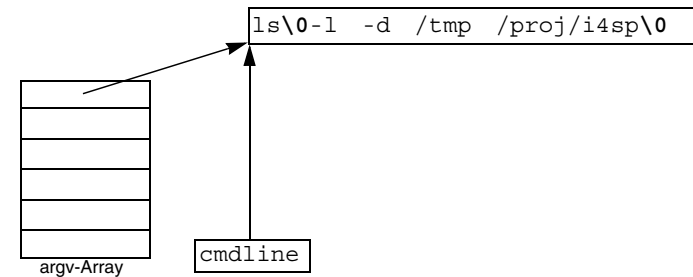
- Wird sukzessive aufgerufen und liefert jeweils einen Zeiger auf das nächste Token (mehrere aufeinanderfolgende Trennzeichen werden hierbei übersprungen)
 - ◆ *str* ist im ersten Aufruf ein Zeiger auf den zu teilenden String, in allen Folgeaufrufen NULL
 - ◆ *delim* ist ein String, der alle Trennzeichen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



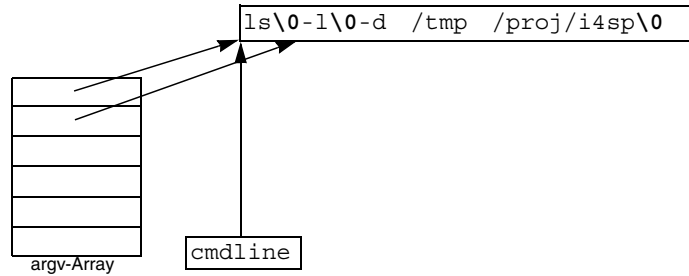
- Kommandozeile befindet sich als '\0'-terminierter String im Speicher

2 strtok-Beispiel



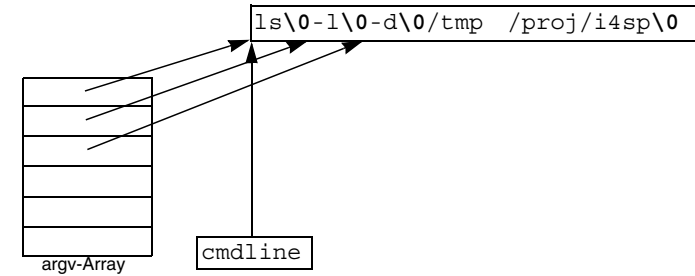
- Erster **strtok**-Aufruf mit dem Zeiger auf diesen Speicherbereich
- **strtok** liefert Zeiger auf erstes Token *ls* und ersetzt den Folgetrenner mit '\0'

2 strtok-Beispiel



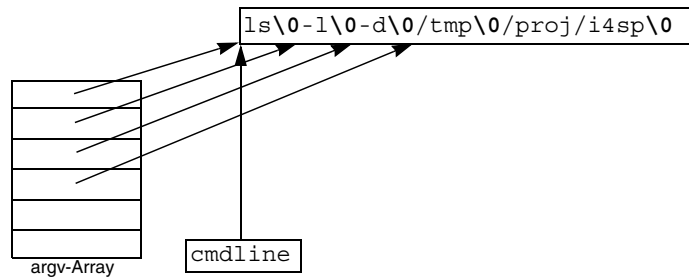
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

2 strtok-Beispiel



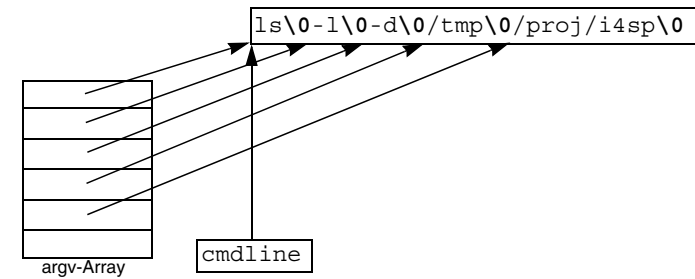
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

2 strtok-Beispiel



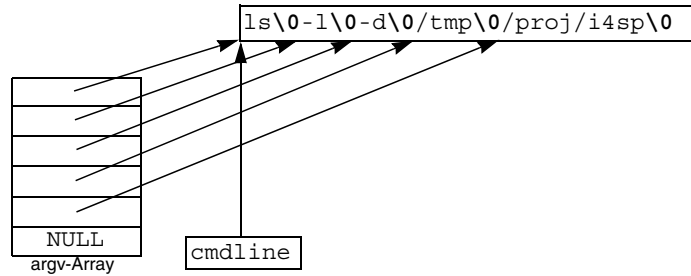
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

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`