

U5 5. Übung

- Aufgabenbesprechung
 - ◆ Aufgabe 2: wsort
 - ◆ Heap- vs. Stackallokation
- Besprechung der 3. Aufgabe halde
- Prozesse
 - ◆ Speicheraufbau
 - ◆ Systemschnittstelle: fork(2), exec(3), exit(3), wait(2), waitpid(2)
- Aufgabe 3: clash (Einfache Shell im Eigenbau)
 - ◆ Ziele der Aufgabe
 - ◆ Funktionsprinzip
 - ◆ String-Stückelung mit **strtok(3)**
 - ◆ Ermitteln von Systemlimits mit sysconf(3)

U5-1 Aufgabe 2: Sortieren mit qsort

1 wsort - Datenstrukturen (1. Möglichkeit)

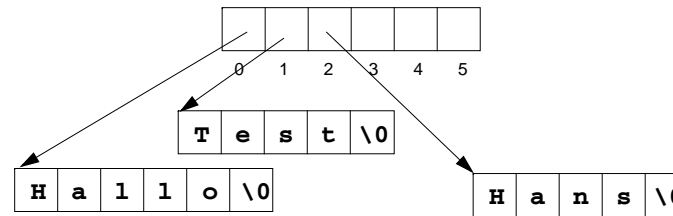
- Array von Zeichenketten (Größe: Anzahl der Wörter * 101 * sizeof(char))

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

- Vorteile:
 - ◆ einfach (z.B. Speicherfreigabe durch Freigeben des Feldes)
- Nachteile:
 - ◆ hoher Kopieraufwand (303 Bytes pro Umordnung)
 - ◆ maximale Länge der Wörter muss bekannt sein
 - ◆ Verschwendung von Speicherplatz
 - ◆ Reallokation teuer, da eventuell alle Daten kopiert werden müssen

2 wsort - Datenstrukturen (2. Möglichkeit)

- Array von Zeigern auf Zeichenketten (Größe: Anzahl der Wörter * sizeof(char*))



- Vorteile:
 - ◆ schnell, da nur Zeiger vertauscht werden (x86-32: 12 Bytes pro Umordnung)
 - ◆ Zeichenketten können beliebig lang sein
 - ◆ sparsame Speichernutzung
- Nachteil:
 - ◆ Freigabe komplizierter: zuerst Wörter, dann Zeiger-Array freigeben

U5-2 Heap- vs. Stackallokation

- Beispiel mit 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 Schreiarbeit
 - ◆ verschlechtert Code-Lesbarkeit
 - ◆ zeitaufwendig (relevant z.B. in der Klausur)

U5-2 Heap- vs. Stackallokation

- Alternative: (dynamische) Stackallokation

```
char buffer[102];

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

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

U5-3 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;
    return(EXIT_SUCCESS);
}
```

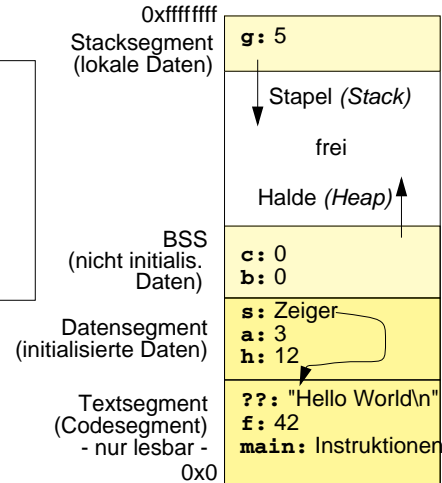
- ◆ Vergleiche Vorlesung: A / V Vom Programm zum Prozess, Seite 7f.

U5-4 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;
    return(EXIT_SUCCESS);
}
```



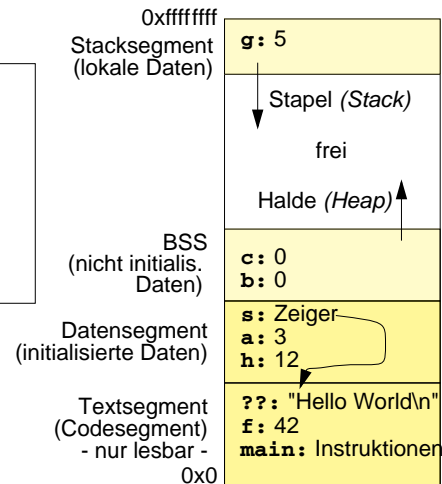
U5-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

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

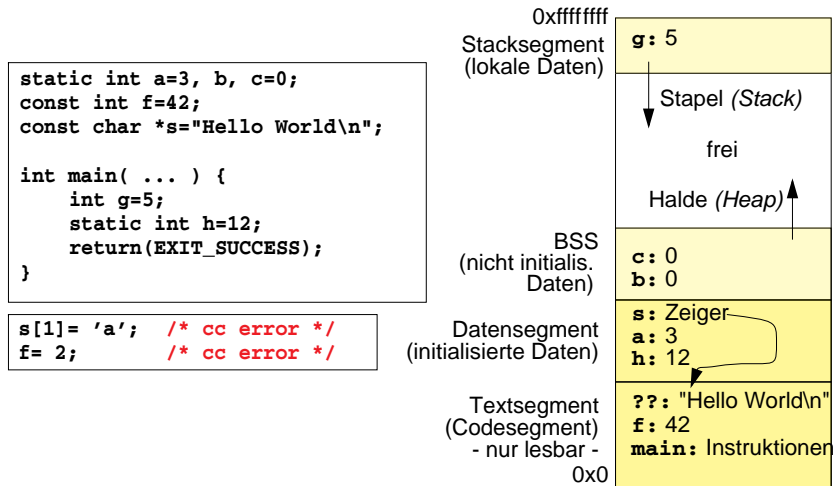
int main( ... ) {
    int g=5;
    static int h=12;
    return(EXIT_SUCCESS);
}
```

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



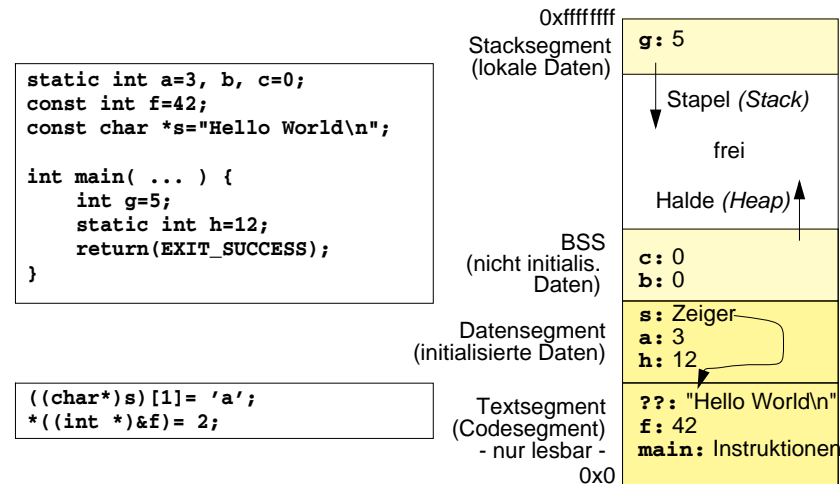
U5-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



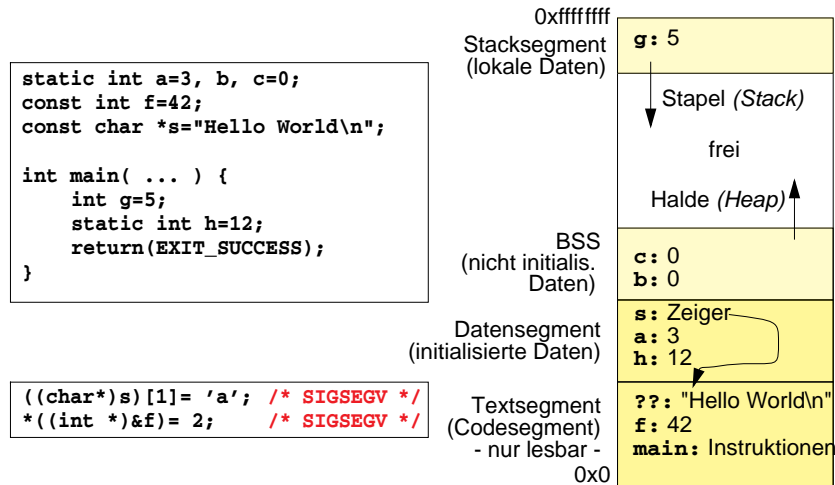
U5-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



U5-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

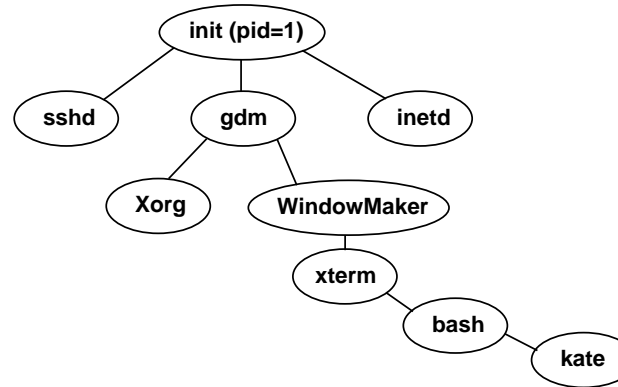


U5-5 Prozesse: Überblick

- Prozesse sind eine Ausführungsumgebung für Programme (Vorlesung A | III-3)
 - ◆ haben eine Prozess-ID (PID, ganzzahlig positiv)
 - ◆ führen ein Programm aus
- Mit einem Prozess sind Ressourcen verknüpft, z.B.
 - ◆ Speicher
 - ◆ Adressraum
 - ◆ offene Dateien

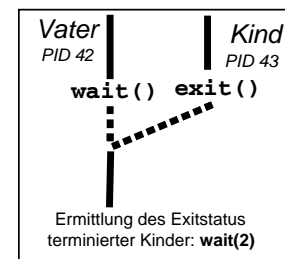
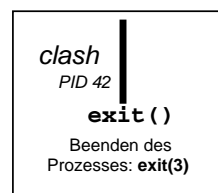
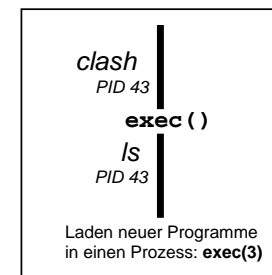
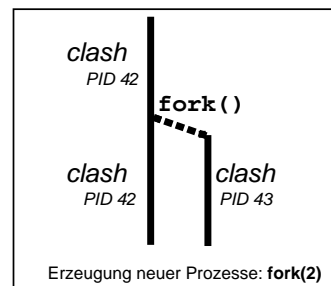
U5-5 UNIX-Prozesshierarchie

- 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**

U5-6 POSIX-Prozess-Systemfunktionen

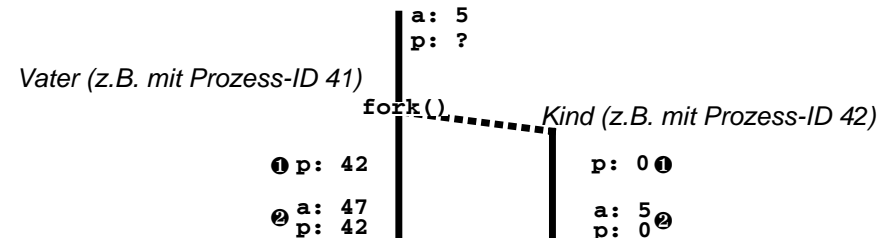


1 fork(2): Erzeugung eines neuen Prozesses

- Erzeugt einen neuen Kindprozess (Vorlesung A | III-5)
- 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 (Vorl. A | III-5.3)
- **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"`)

■ 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 exec1(const char *path, const char *arg0, ...
        /*, (char *) NULL */);
```

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

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

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

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

- Anmerkung: Alle Varianten von `exec(2)` erwarten als letzten Eintrag in der Argumentenliste einen `NULL`-Zeiger.

3 exit(3)

- beendet aktuellen Prozess mit einem Status-Byte
 - ◆ Konvention: Status 0 bedeutet Erfolg, alles andere eine Fehlernummer
 - Exitstatus **EXIT_FAILURE** und **EXIT_SUCCESS** vordefiniert
 - ◆ Bedeutung der Exitstatus üblicherweise in Manpage dokumentiert
- gibt alle Ressourcen frei, die der Prozess belegt hat, z.B.
 - ◆ Speicher
 - ◆ Filedeskriptoren (schließt alle offenen Dateien)
 - ◆ 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 Ressourcen und sollten zeitnah 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
        // Zur Ausgabe des Statuses siehe Makros in wait(2)

    } 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 {
        // Fehler bei fork
        ...
    }
}
```

4 wait(2)

- **wait** blockiert, bis ein Kind-Prozess terminiert wird
 - ◆ *pid* dieses Kind-Prozesses wird als Rückgabewert geliefert
 - ◆ als Parameter kann ein Zeiger auf einen *int*-Wert mitgegeben werden, in dem unter anderem der Exitstatus 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)**
 - Exitstatus: **WEXITSTATUS(status)**
 - weitere siehe **wait(2)**

5 waitpid(2)

- Mächtigere Variante von **wait(2)**

```
pid_t waitpid(pid_t pid, int *status, int options);
```
- Wartet auf Statusänderung eines
 - ◆ *bestimmten* Prozesses: `pid > 0`
 - ◆ *beliebigen* Kindprozesses: `pid == -1`
- Verhalten mit *Optionen* anpassbar
 - ◆ **WNOHANG**: **waitpid** kehrt sofort zurück, wenn kein passender Zombie verfügbar ist
 - eignet sich zum Polling nach Zombieprozessen

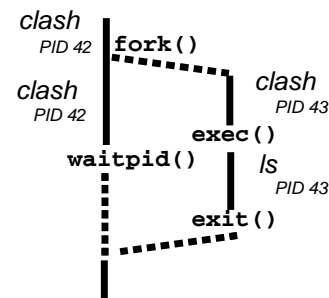
U5-7 Aufgabe 4: Einfache Shell im Eigenbau

1 Ziel der Aufgabe

- Arbeiten mit dem UNIX-Prozesskonzept
- Verstehen von Quellcode anderer Personen (`p11st.c`)
- Erstellen eines Makefiles

2 Funktionsweise

- Eingabezeile, aus der der Benutzer Programme starten kann



- Erzeugt einen neuen Prozess und startet in diesem das Programm
- Vordergrundprozess: Wartet auf die Beendigung des Prozesses und gibt anschließend dessen Exitstatus aus
- Hintergrundprozess: Wartet **nicht** auf Beendigung des Prozesses

3 Aufteilung der Kommandozeile

- Anzahl der Kommandoparameter
 - ◆ 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 zur Laufzeit konstruiert werden
 - ◆ hierzu muss die Kommandozeile 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 **strtok(3)** benutzt werden

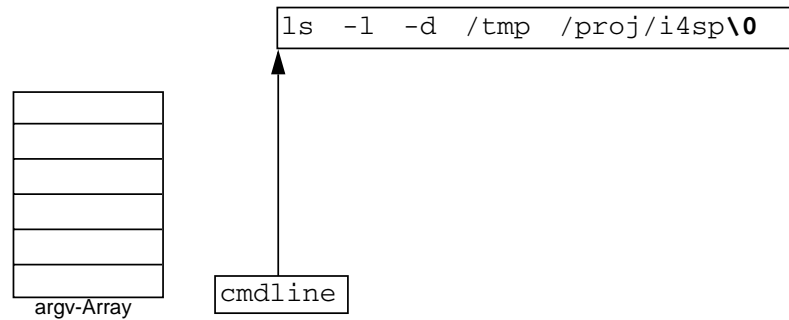
3 strtok

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

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

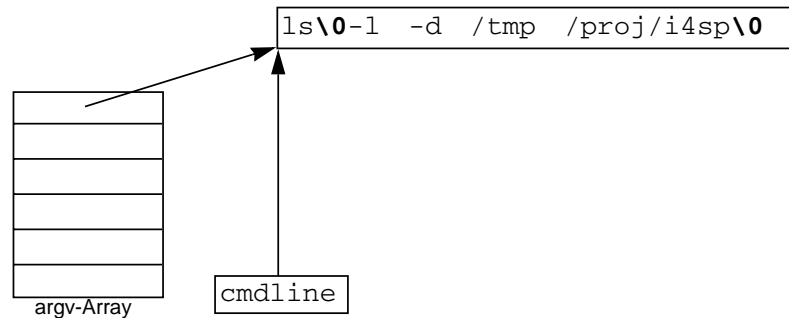
- Wird sukzessive aufgerufen und liefert jeweils einen Zeiger auf das nächste Token (mehrere aufeinanderfolgende Trennzeichen werden hierbei übersprungen)
 - ◆ `str` ist im ersten Aufruf ein Zeiger auf den zu teilenden String, in allen Folgeaufrufen `NULL`
 - ◆ `delim` ist ein String, der alle Trennzeichen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

3 strtok-Beispiel



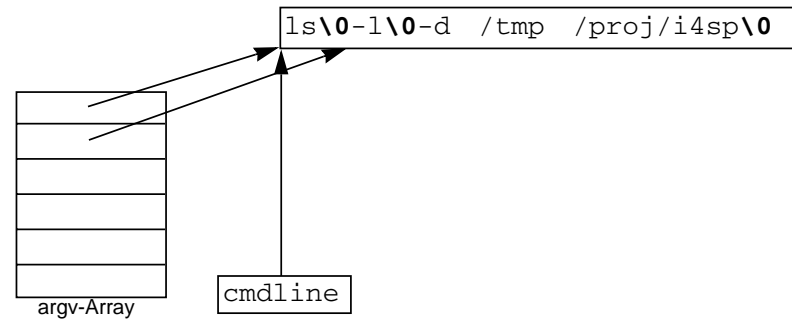
- Kommandozeile befindet sich als '`\0`'-terminierter String im Speicher

3 strtok-Beispiel



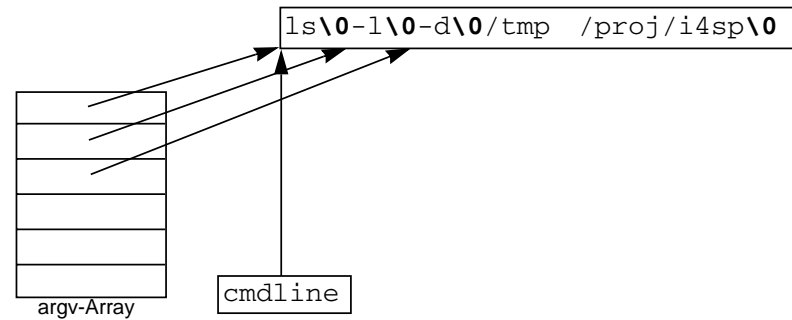
- Erster **strtok**-Aufruf mit dem Zeiger auf diesen Speicherbereich
- **strtok** liefert Zeiger auf erstes Token `/s` und ersetzt den Folgetrenner mit '`\0`'

3 strtok-Beispiel



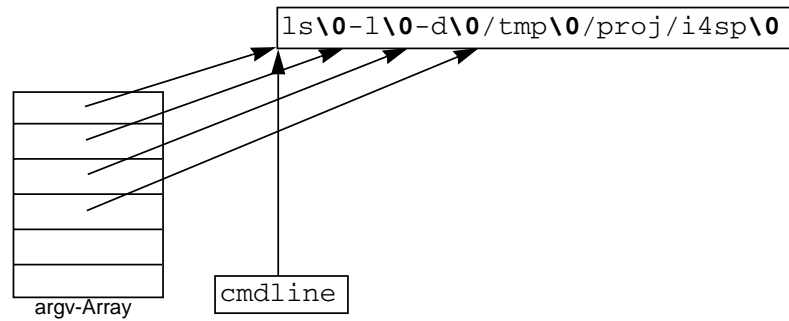
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

3 strtok-Beispiel



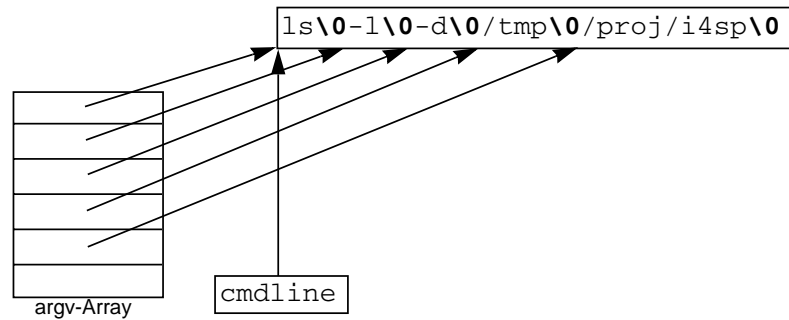
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

3 strtok-Beispiel



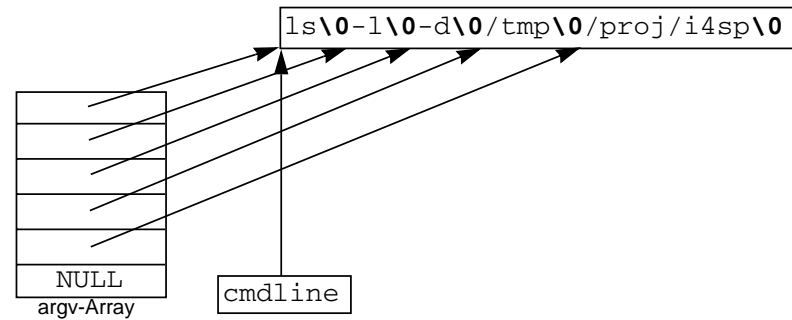
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

3 strtok-Beispiel



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

3 strtok-Beispiel



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

4 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 Zeichenkette, die auf einmal eingelesen werden kann (**stdin** oder Datei)