

- Aufgabenbesprechung
 - ◆ Aufgabe 1: lilo
 - ◆ Fehlerbehandlung
 - ◆ Heap- vs. Stackallokation
- Abgabesystem: Partnerabgabe
- Prozesse
 - ◆ Speicheraufbau
 - ◆ Systemschnittstelle: fork(2), exec(3), exit(3), wait(2), waitpid(2)
- Aufgabe 3: clash (Einfache Shell im Eigenbau)
 - ◆ Funktionsprinzip
 - ◆ String-Stückelung mit **strtok(3)**
 - ◆ Ermitteln von Systemlimits mit sysconf(3)

1 Fehler in Bibliotheksfunktionen

- Fehler treten häufig in Funktionen der C-Bibliothek auf
 - erkennbar i.d.R. am Rückgabewert (Manpage, Sektion **RETURN VALUES**)
- Die Fehlerursache wird über die globale Variable **errno** übermittelt
 - Fehlercode für jeden möglichen Fehler (siehe **errno(3)**)
 - Der Wert **errno=0** ist reserviert, alles andere ist ein Fehlercode
 - Bibliotheksfunktionen setzen **errno** im Fehlerfall (sonst nicht *zwingend*)
 - 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
}
```

- 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: Kopierprogramm: Öffnen der Quelldatei schlägt fehl
 - ➡ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden

2 Ausgabe von Fehlermeldungen

- 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

3 Fehlererkennung - Spezialfunktionen

- 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
- 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
}
```

4 Fehlererkennung - Auswertung von `errno`

- 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

U3-2 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 4)
- erfordert Fehlerbehandlung
- viel Schreibarbeit
 - ◆ verschlechtert Code-Lesbarkeit
 - ◆ zeitaufwendig (z.B. in der Klausur)

U3-2 Heap - vs. Stackallokation

- Alternative: (dynamische) Stackallokation

```
char buffer[102];

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

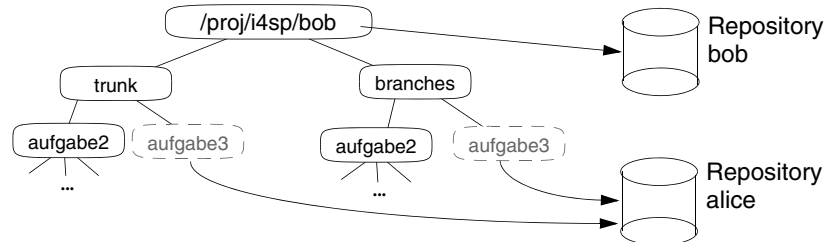
- Sehr effizient
 - ◆ Allokation: `Stackpointer -= 102;`
 - ◆ Freigabe: `Stackpointer += 102;`
- Keine Fehlerbehandlung durch das Programm
 - ◆ Stacküberlauf wird ggf. vom Betriebssystem erkannt (`SIGSEGV`)
- Implizite Freigabe beim Verlassen der Funktion
- Keine Speicherlecks möglich

U3-3 Abgabesystem: Team-Arbeit

- Gemeinsame Bearbeitung im Repository eines Teammitglieds
 - ◆ Repository-Eigentümer: *alice*
 - ◆ Partner (nutzt Repository von *alice*): *bob*
- Abgabe erfolgt ebenfalls im Repository des Eigentümers
 - ◆ es ist nur eine Abgabe erforderlich
- **Hinweis:** bei Verständnis-Problemen zu Subversion empfiehlt sich die Lektüre zumindest der ersten beiden Kapitel des SVN-Buchs
 - ◆ <http://svnbook.red-bean.com/>
- Machen Sie sich frühzeitig mit dem Bearbeitungs-/Abgabeprozess vertraut
 - ◆ Arbeiten Sie von Beginn an in Ihrem Projektverzeichnis
 - ◆ Checken Sie auch Zwischenstände Ihrer Bearbeitung in der Repository ein
 - ◆ Sie können zu Beginn auch leere Dateien einchecken und abgeben
 - ◆ selbstverschuldetete verspätete Abgaben werden nicht angenommen!

2 Ablauf für den Partner

- Partner setzt in seinem Repository einen Verweis auf Hauptrepository
 - ◆ technisch: svn:external-Property
 - ◆ irrelevant für Abgabe, unterstützt nur den Prozess der Teamarbeit
 - ◆ Achtung: Abgabe im eigenen Repository überlagert Partnerabgabe
 - ➔ zum Umstieg auf Teamarbeit eigene Abgabe löschen (Übungsleiter hilft)



- Arbeit und Abgabe in der eigenen Arbeitskopie normal möglich

1 Ablauf für den Hauptbearbeiter

- Der Partner wird für jede Team-Aufgabe separat festgelegt

```
alice@fau101$ /proj/i4sp/bin/set-partner aufgabe3 bob
```

- Hintergrund

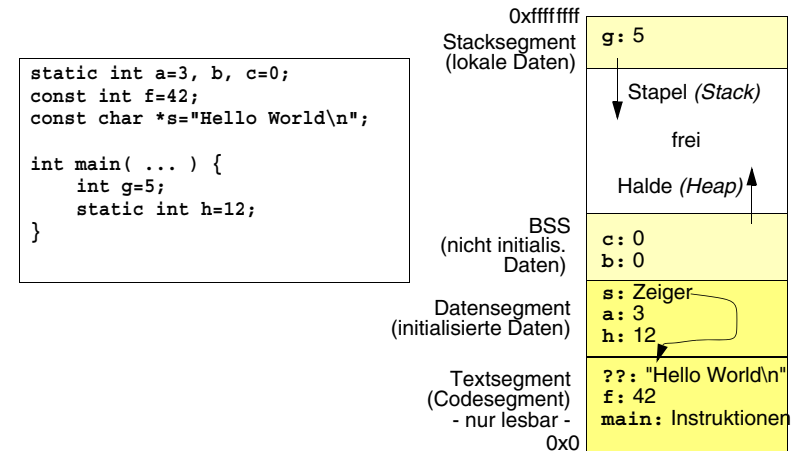
- ◆ Erzeugung und Commit einer Textdatei *partner* in *trunk/aufgabe3*
- ◆ diese Datei enthält den Login-Namen (*bob*) des Partners für diese Aufgabe
- ◆ Partner erhält Zugriff auf die relevanten Teile des Repositories
 - *trunk/aufgabe3*
 - *branches/aufgabe3*

- Abgabe wie gewohnt

```
alice@fau101$ /proj/i4sp/bin/submit aufgabe3
```

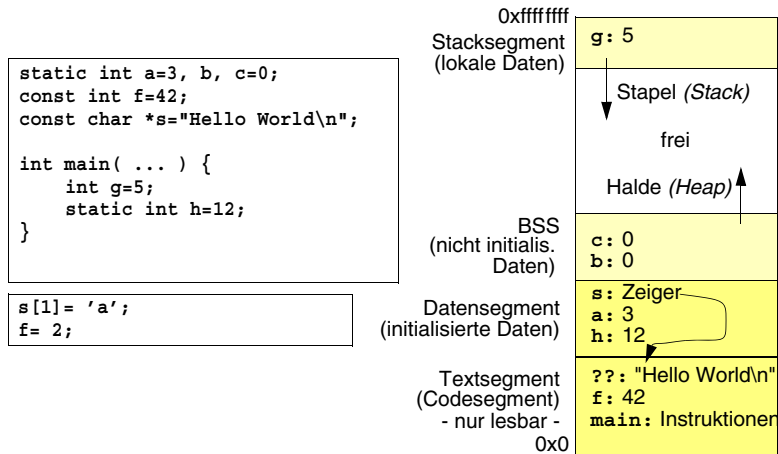
U3-4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente



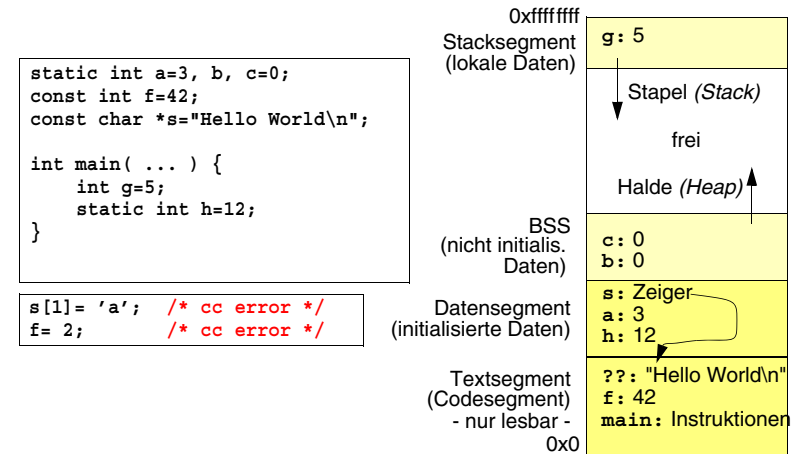
U3-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



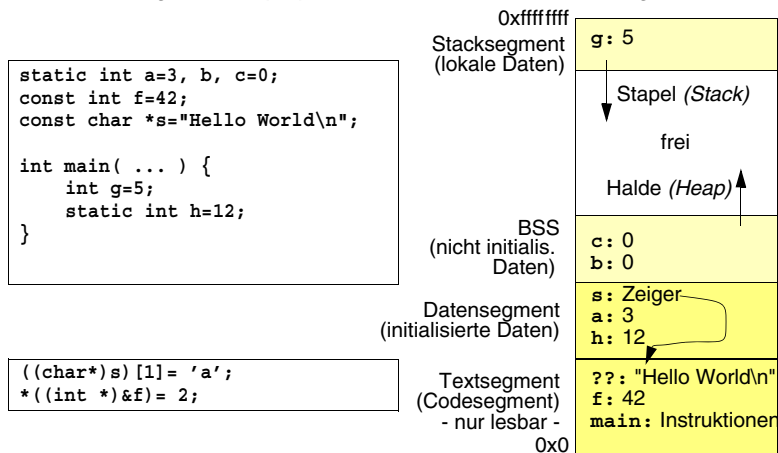
U3-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



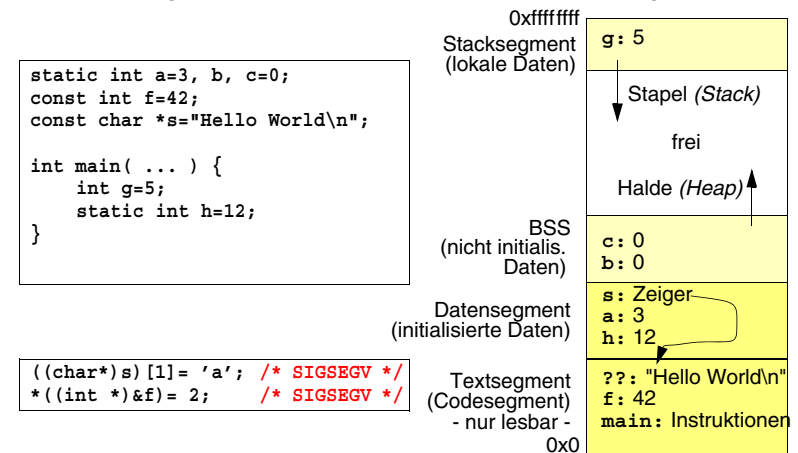
U3-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



U3-4 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

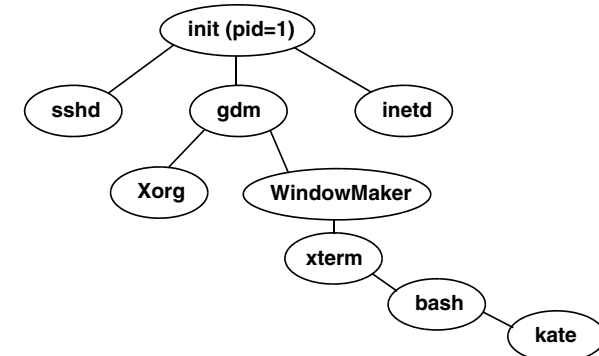


U3-5 Prozesse: Überblick

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

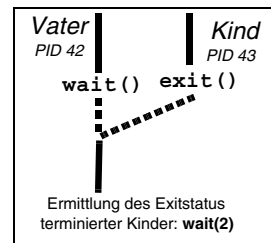
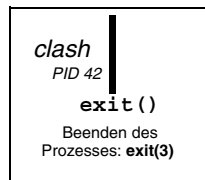
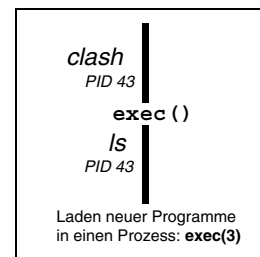
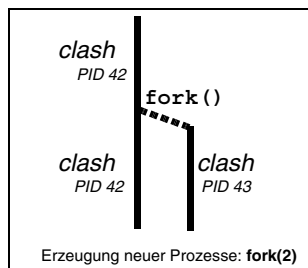
U3-5 UNIX-Prozesshierarchie

- 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**

U3-6 POSIX Prozess-Systemfunktionen

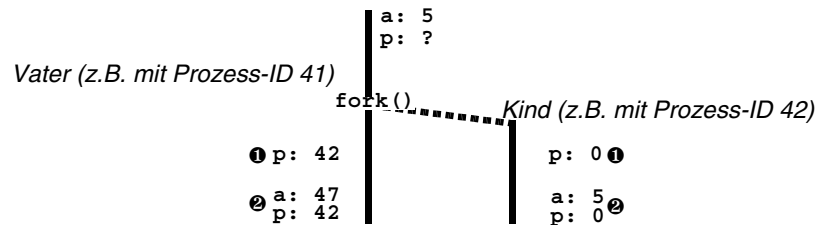


1 fork(2): Erzeugung eines neuen Prozesses

- 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, ...
    /*, (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, ...
    /*, (char *) NULL */);
```

```
int execvp(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: Filedescriptoren (= 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
 - ◆ Filedescriptoren (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 {                // Fehler bei fork
        ...
    }
}
```

4 wait(2)

- `wait` blockiert, bis ein Kind-Prozess terminiert oder gestoppt wird
 - ◆ `pid` dieses Kind-Prozesses wird als Rückgabewert 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

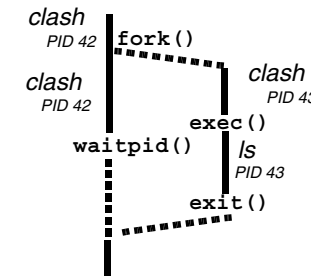
5 waitpid(2)

- 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

U3-7 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

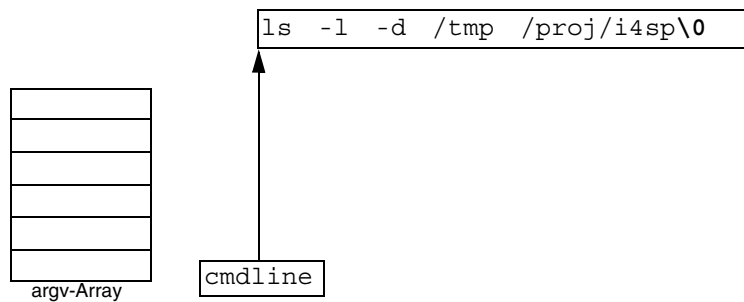
- Anzahl der Kommandoparameter beim Programmieren
 - gibt der Benutzer mit der Eingabe vor
 - können von Kommando zu Kommando unterschiedlich sein
 - die l-Varianten von exec können nicht verwendet werden
- Die v-Varianten von exec erhalten ein Argumentenarray als Parameter
 - dieses kann zur Laufzeit konstruiert werden
 - hierzu muss die Kommandozeile 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 **strtok(3)** benutzt werden

2 strtok

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

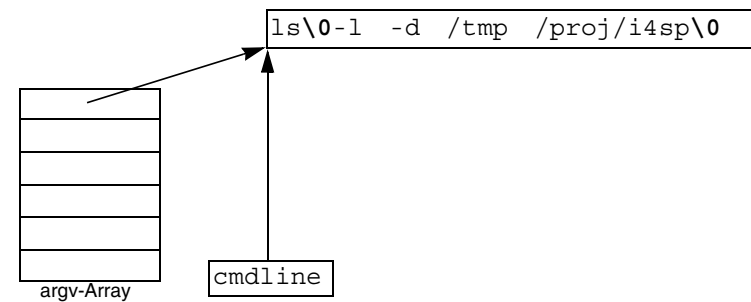

```
char *strtok(char *str, const char *delim);
```
- Wird sukzessive aufgerufen und liefert jeweils einen Zeiger auf das nächste Token (mehrere aufeinanderfolgende Trennzeichen werden hierbei übersprungen)
 - ◆ `str` ist im ersten Aufruf ein Zeiger auf den zu teilenden String, in allen Folgeaufrufen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



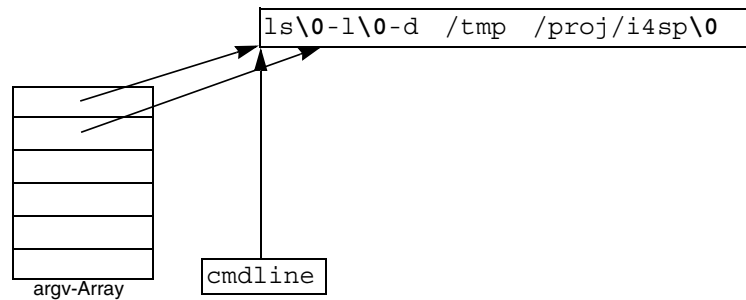
- Kommandozeile befindet sich als '\0'-terminierter String im Speicher

2 strtok-Beispiel



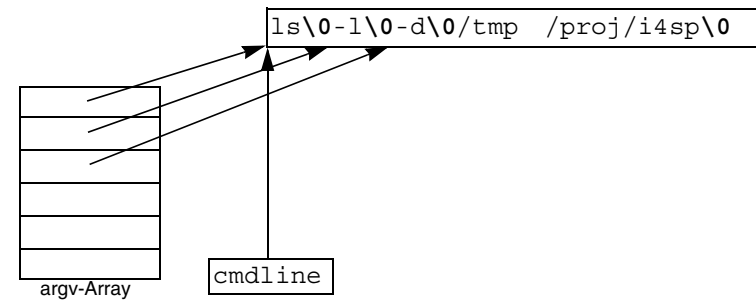
- Erster **strtok**-Aufruf mit dem Zeiger auf diesen Speicherbereich
- **strtok** liefert Zeiger auf erstes Token `ls` und ersetzt den Folgetrenner mit '\0'

2 strtok-Beispiel



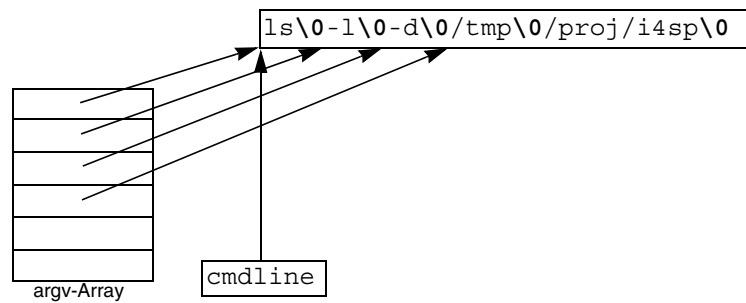
- Weitere Aufrufe von **strtok** nun mit einem NULL-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

2 strtok-Beispiel



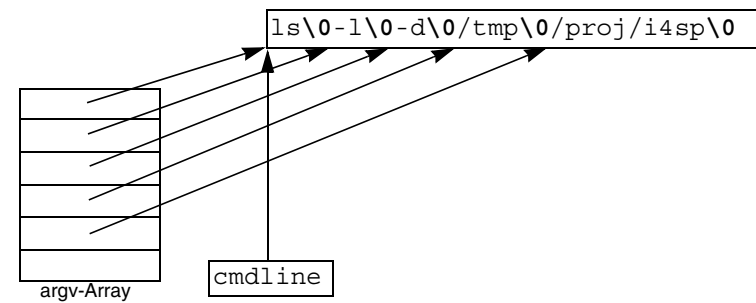
- Weitere Aufrufe von **strtok** nun mit einem NULL-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

2 strtok-Beispiel



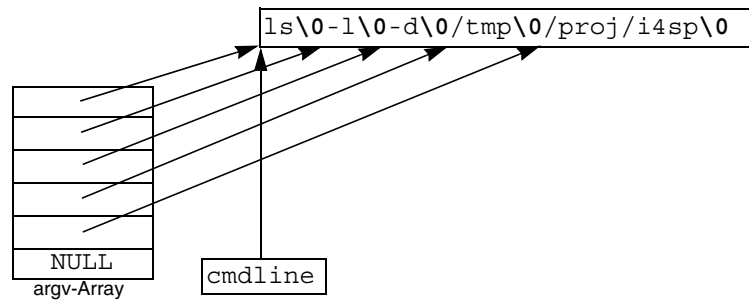
- Weitere Aufrufe von **strtok** nun mit einem NULL-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

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

U3.37

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 (`limits.h`)
 - ◆ `ARG_MAX`
 - ◆ `LINE_MAX`

U3.38