

Übungen zu Systemprogrammierung 1 (SP1)

VL 5 – Prozesse

Jens Schedel, Christoph Erhardt, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2012/13 – 03. Dezember bis 07. Dezember 2012

http://www4.cs.fau.de/Lehre/WS12/V_SP1

Agenda

- 5.1 Adressraumstruktur
- 5.2 Prozesse
- 5.3 Aufgabe 4: clash
- 5.4 Gelerntes Anwenden



5.1 Adressraumstruktur

5.2 Prozesse

5.3 Aufgabe 4: clash

5.4 Gelerntes Anwenden



Aufteilung des Adressraums

- 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);
}
```

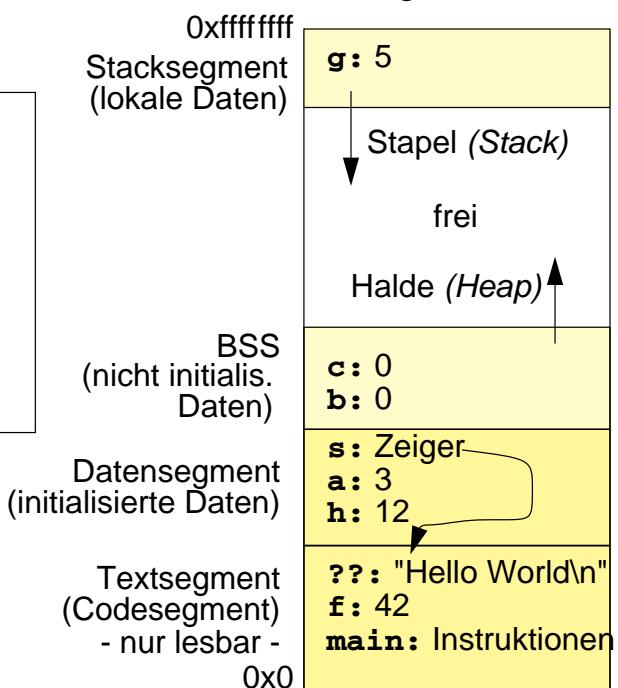
- ◆ Vergleiche Vorlesung: A / V Vom Programm zum Prozess, Seite 7f.



```
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);
}
```

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente



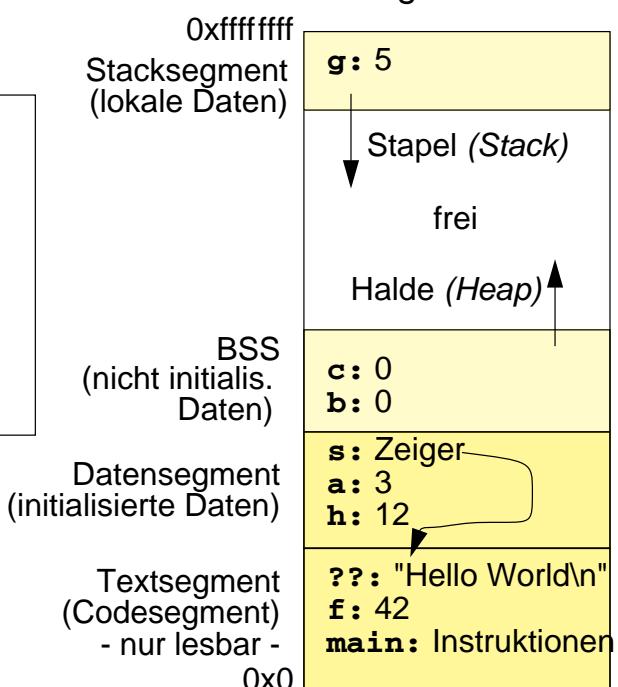
Aufteilung des Adressraums

■ 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;
```



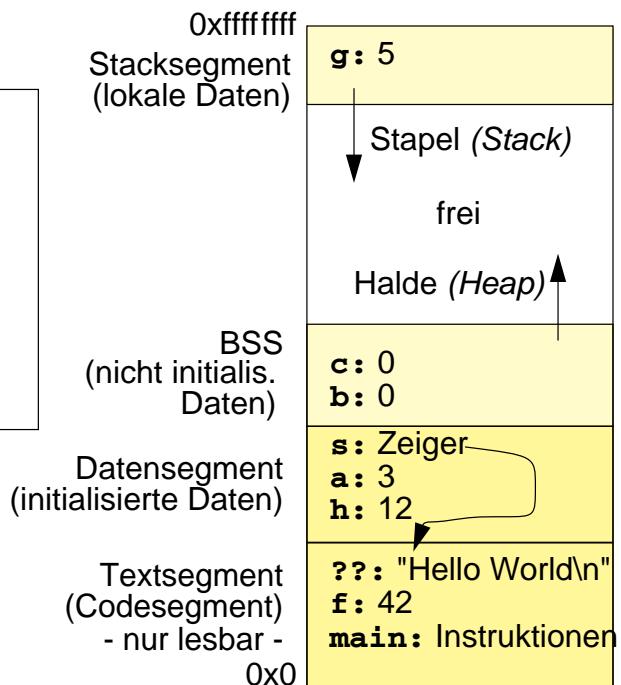
Aufteilung des Adressraums

■ 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'; /* cc error */
f= 2; /* cc error */
```



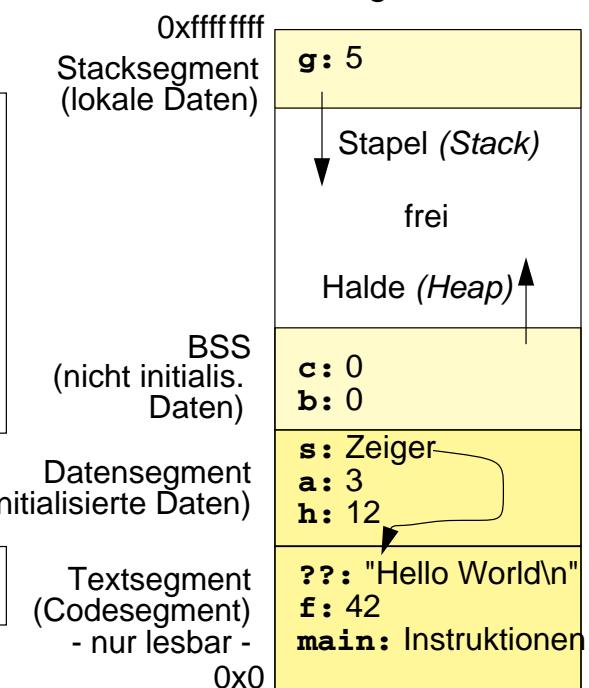
Aufteilung des Adressraums

■ 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);
}

((char*)s)[1]= 'a';
*((int *)&f)= 2;
```

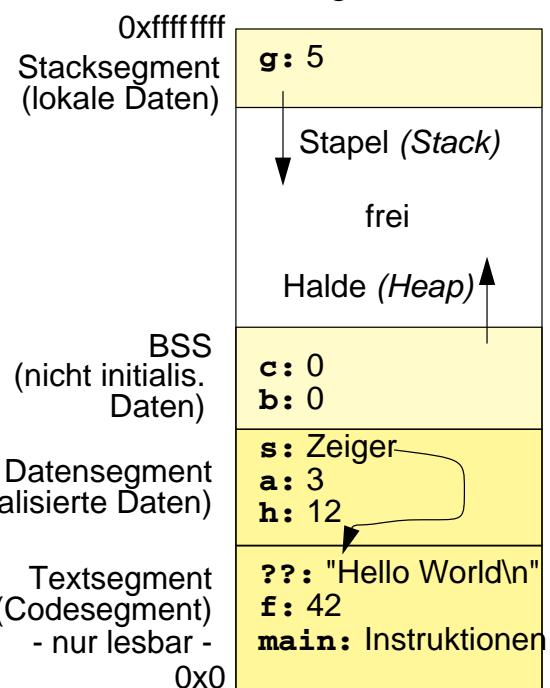


■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
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);
}

((char*)s)[1]= 'a'; /* SIGSEGV */
*((int *)&f)= 2;    /* SIGSEGV */
```



Agenda

5.1 Adressraumstruktur

5.2 Prozesse

5.3 Aufgabe 4: clash

5.4 Gelerntes Anwenden

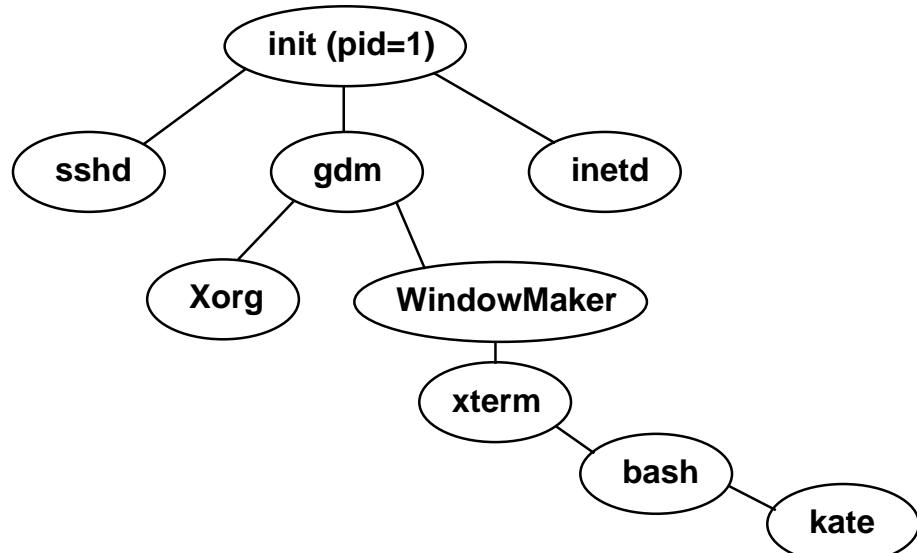


- Prozesse sind eine Ausführumgebung für Programme (vgl. Vorlesung A | III-3)
 - haben eine Prozess-ID (PID, ganzzahlig positiv)
 - führen ein Programm aus
- Mit einem Prozess sind Ressourcen verknüpft, z.B.
 - Speicher
 - Adressraum
 - offene Dateien



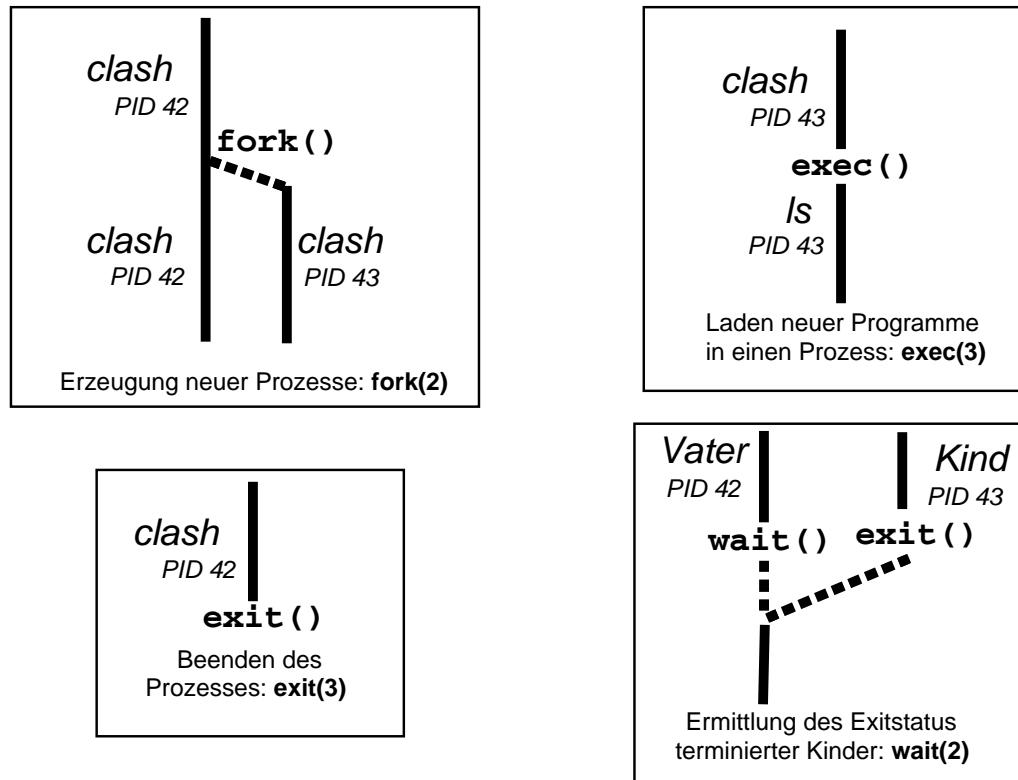
Prozessehierarchie

- 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**





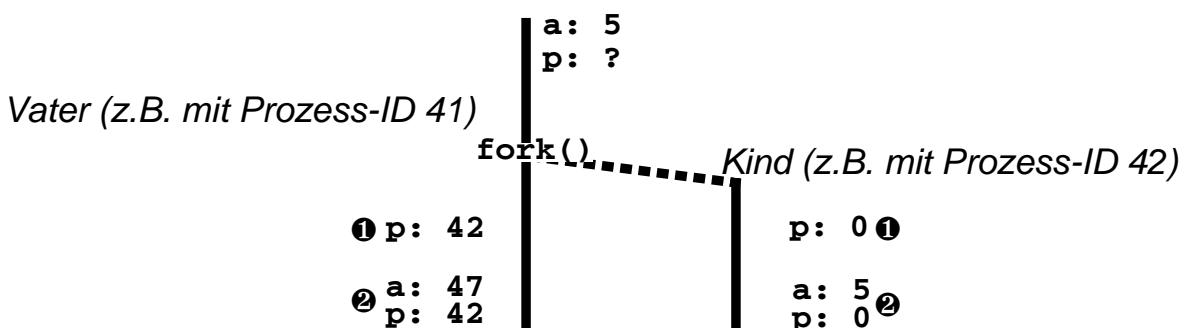
fork(2)

- 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(2)) entscheiden, ob es sich um den Vater- oder den Kindprozess handelt



fork(2)

```
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
    ...
}
```



exec(2)

- Lädt Programm zur Ausführung in den aktuellen Prozess (vgl. Vorlesung A | III-5.3)
- ersetzt aktuell ausgeführtes Programm: Text-, Daten- und Stacksegment
- behält: Filedeskriptoren (= geöffnete Dateien), Arbeitsverzeichnis, ...
- Aufrufparameter:
 - Dateiname des neuen Programmes
 - Argumente, die der main-Funktion des neuen Programms übergeben werden
- Beispiel

```
execl("/bin/cp", "/bin/cp", "/etc/passwd", "/tmp/passwd", NULL);
```
- `exec` kehrt nur im Fehlerfall zurück



- 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[]);
```

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



exit(2)

- beendet aktuellen Prozess mit angegebenem Exitstatus
- 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
 - 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



wait(2)

- Warten auf Statusinformationen von Kind-Prozessen mit Hilfe von `wait(2)`
- 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  
        ...  
    }  
}
```



wait(2)

- `wait(2)` 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(2)` terminiert: `WIFEXITED(status)`
 - Exitstatus: `WEXITSTATUS(status)`
 - weitere siehe `wait(2)`



- Mächtigere Variante von `wait(2)`
- Wartet auf Statusänderung eines
 - bestimmten Prozesses: `pid>0`
 - beliebigen Kindprozesses: `pid== -1`
- Verhalten mit Optionen anpassbar
 - `WNOHANG`: `waitpid(2)` kehrt sofort zurück, wenn kein passender Zombie verfügbar ist
 - eignet sich zum Polling nach Zombieprozessen



Agenda

- 5.1 Adressraumstruktur
- 5.2 Prozesse
- 5.3 Aufgabe 4: clash
- 5.4 Gelerntes Anwenden

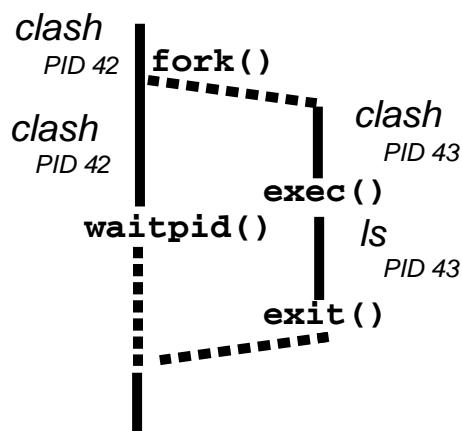


- Arbeiten mit dem UNIX-Prozesskonzept
- Verstehen von Quellcode anderer Personen (plist.c)
- Erstellen eines Makefiles mit mehreren Dateien



Funktionsweise der clash

- 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



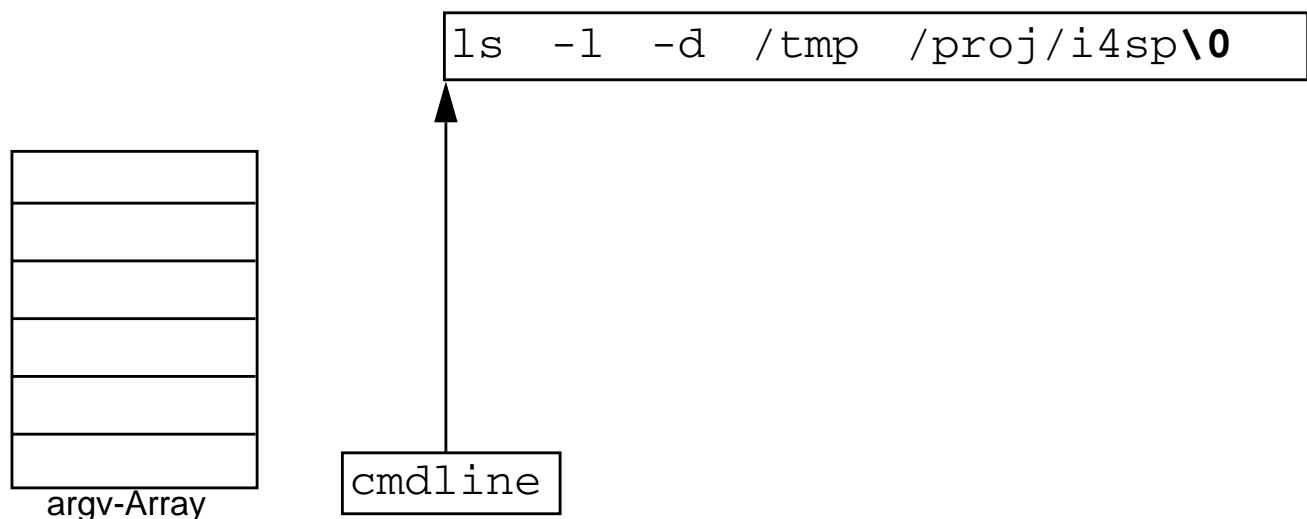
- Anzahl der Kommandoparameter
 - gibt der Benutzer mit der Eingabe vor
 - können von Kommando zu Kommando unterschiedlich sein
 - die l-Varianten von exec(2) können nicht verwendet werden
- Die v-Varianten von exec(2) erhalten ein Argumentenarray als Parameter
 - dieses kann zur Laufzeit konstruiert werden
 - hierzu muss die Kommandozeile aufgeteilt werden (Trenner '\t' und ' ')
 - das Argumentenarray ist ein Feld von Zeigern auf die einzelnen Token
 - terminiert mit einem *NUL*-Zeiger
- Zum Aufteilen der Kommandozeile kann strtok(3) benutzt werden



strtok(3)

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

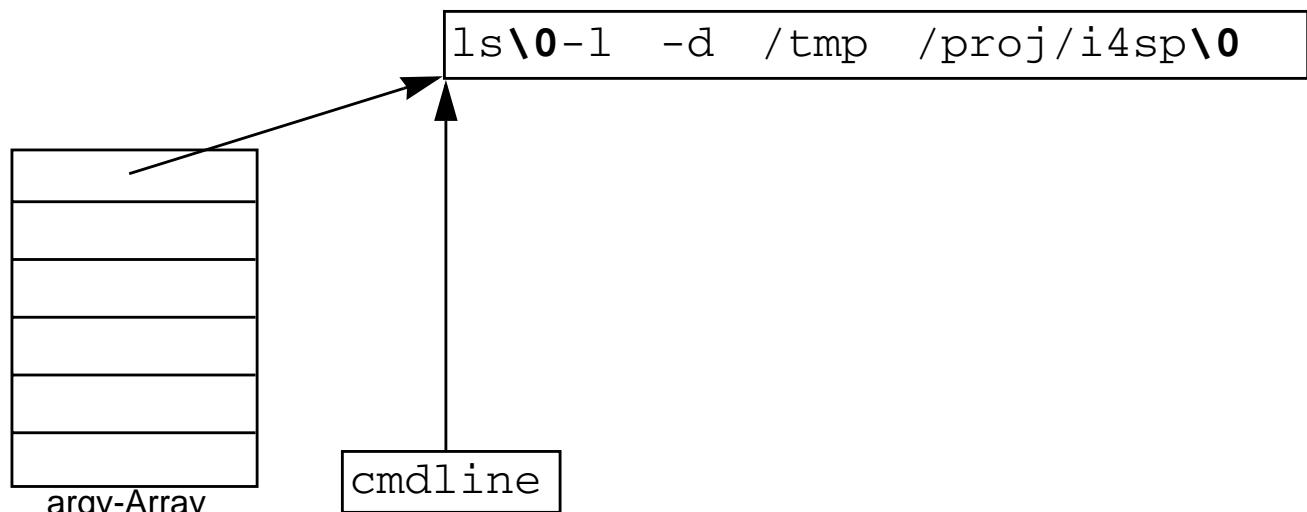




- Kommandozeile befindet sich als \0-terminierter String im Speicher

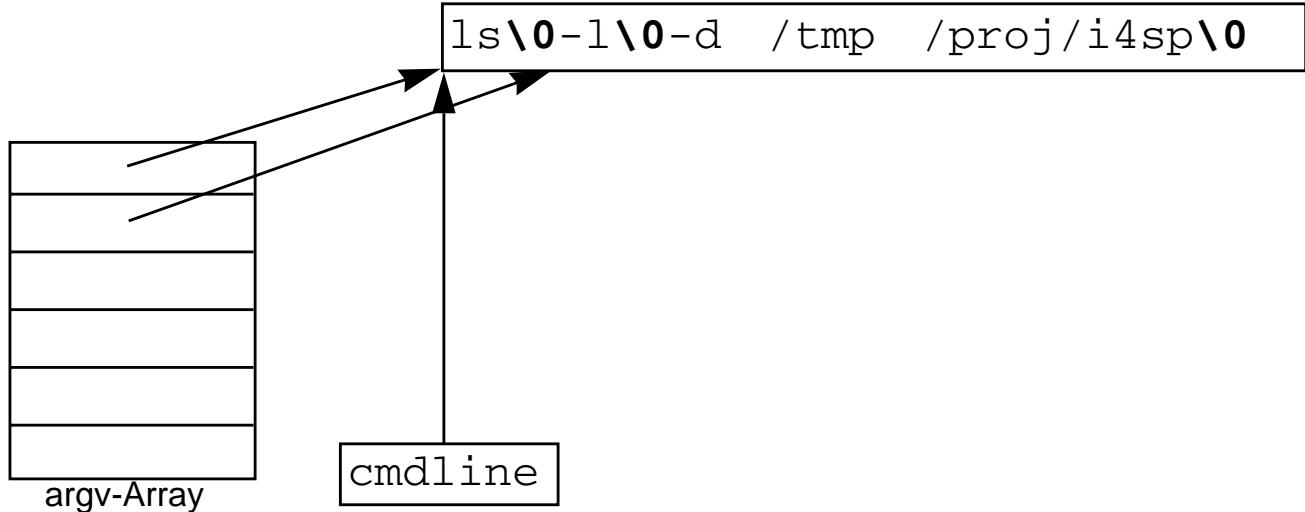


strtok(3)



- Erster `strtok(3)`-Aufruf mit dem Zeiger auf diesen Speicherbereich
- `strtok(3)` liefert Zeiger auf erstes Token `ls` und ersetzt den Folgetrenner mit '`\0`'

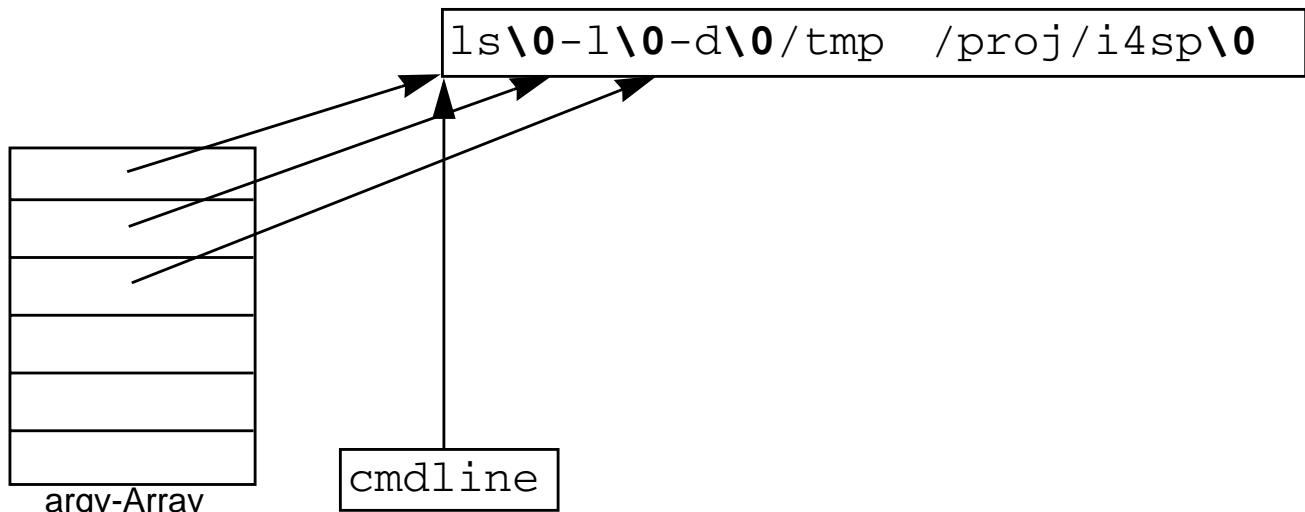




- Weitere Aufrufe von **strtok(3)** nun mit einem *NULL*-Zeiger
- **strtok(3)** liefert jeweils Zeiger auf das nächste Token

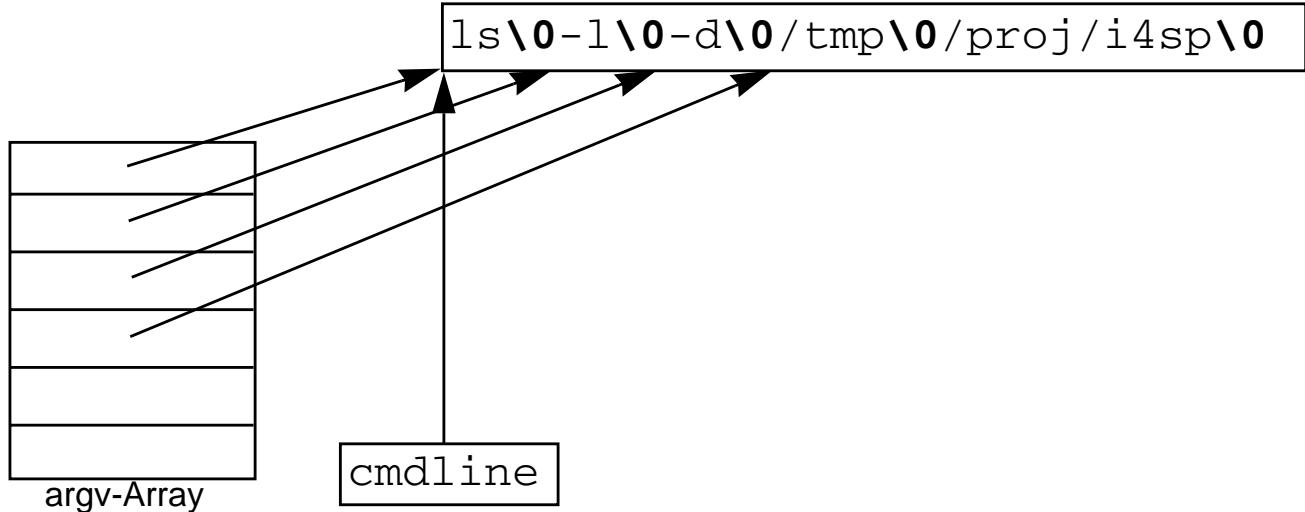


strtok(3)



- Weitere Aufrufe von **strtok(3)** nun mit einem *NULL*-Zeiger
- **strtok(3)** liefert jeweils Zeiger auf das nächste Token

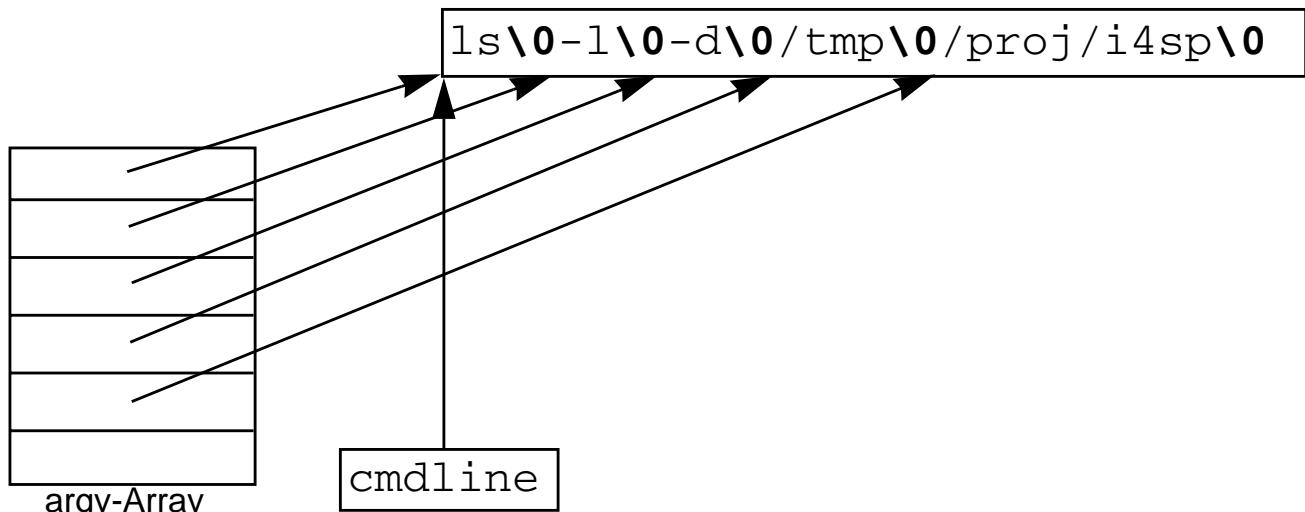




- Weitere Aufrufe von `strtok(3)` nun mit einem *NULL*-Zeiger
- `strtok(3)` liefert jeweils Zeiger auf das nächste Token

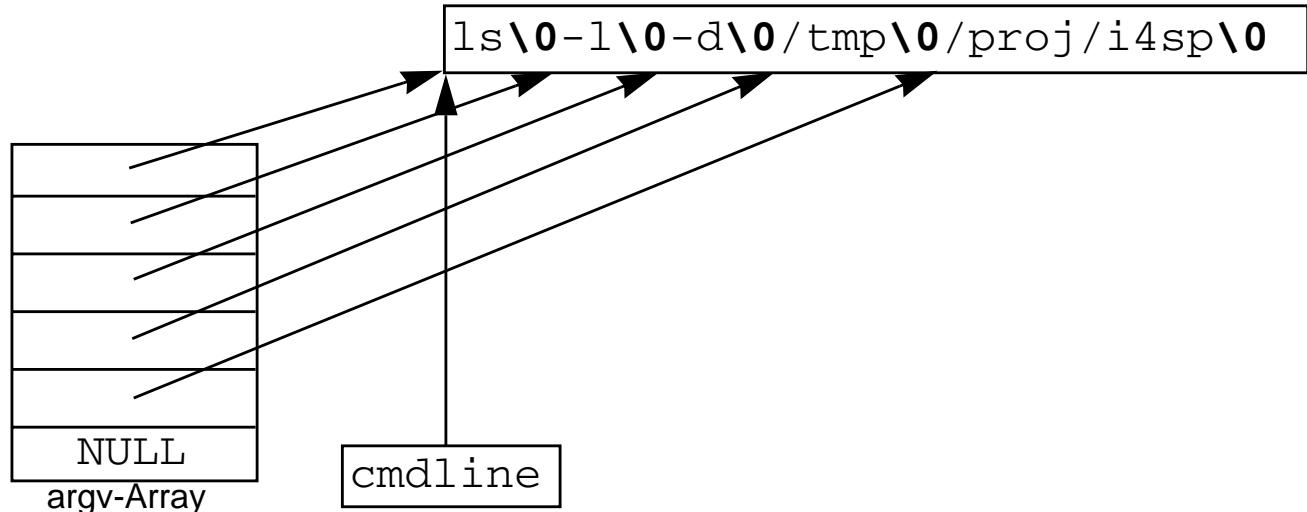


strtok(3)



- Weitere Aufrufe von `strtok(3)` nun mit einem *NULL*-Zeiger
- `strtok(3)` liefert jeweils Zeiger auf das nächste Token





- Weitere Aufrufe von `strtok(3)` nun mit einem *NULL*-Zeiger
- Am Ende liefert `strtok(3)` *NULL* und das argv-Array hat die nötige Form



Funktion sysconf(3)

- Abfrage von Konfigurationsoptionen des Betriebssystems, z.B.
 - Maximale Länge der Kommandozeile für `exec(3)` (`_SC_ARG_MAX`)
 - Maximale Länge einer Zeichenkette, die auf einmal eingelesen werden kann (stdin oder Datei) (`_SC_LINE_MAX`)



- 5.1 Adressraumstruktur
- 5.2 Prozesse
- 5.3 Aufgabe 4: clash
- 5.4 Gelerntes Anwenden



Aktive Mitarbeit!

„Aufgabenstellung“

Programmieraufgabe aus der Miniklausur vom 14. Juni 2010

