U7 POSIX-Prozesse und Signale

- Besprechung Aufgaben 4 und 5
- Fehlerbehandlung
- Prozesse
- POSIX-Prozess-Systemfunktionen
- POSIX-Signale
- Aufgabe 6

U7-1 Fehlerbehandlung

- 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: Öffnen einer zu kopierenden Datei schlägt fehl
    ➠ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden

1 Fehler in Bibliotheksfunktionen

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

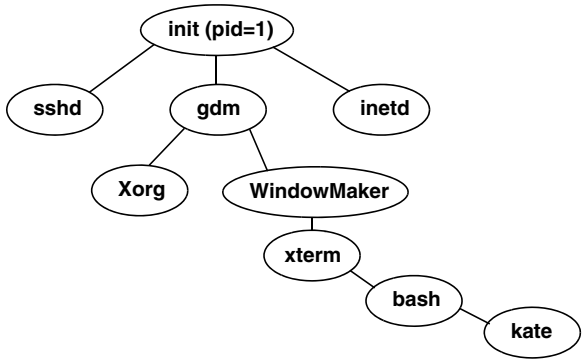
```
char *mem = malloc(...); /* malloc gibt im Fehlerfall */
if(NULL == mem) {        /* NULL zurück */
    fprintf(stderr, "%s:%d: malloc failed with reason: %s\n",
        __FILE__, __LINE__-3, strerror(errno));
    perror("malloc"); /* Alternative zu strerror + fprintf */
    exit(EXIT_FAILURE); /* Programm mit Fehlercode beenden */
}
```


U7-2 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 **E.5**)

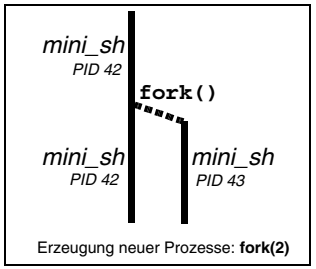
U7-2 UNIX-Prozesshierarchie

- 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

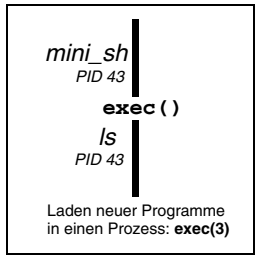


  - Beispiel: **kate** ist ein Kind von **bash**, **bash** wiederum ein Kind von **xterm**

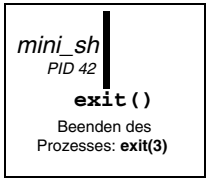
U7-3 POSIX Prozess-Systemfunktionen



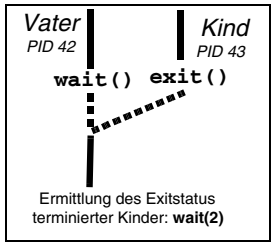
Erzeugung neuer Prozesse: **fork(2)**



Laden neuer Programme in einen Prozess: **exec(3)**



Beenden des Prozesses: **exit(3)**



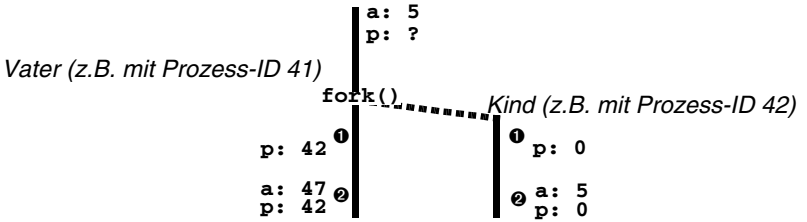
Ermittlung des Exitstatus terminierter Kinder: **wait(2)**

1 fork(2): Erzeugung eines neuen Prozesses

- Erzeugt einen neuen Kindprozesses
- 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
- **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

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;
  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`

U7-4 POSIX-Signale

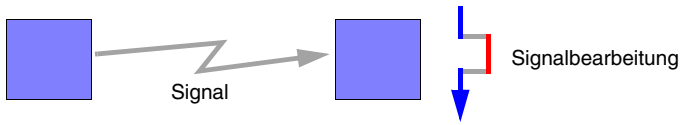
- Zwei Arten von Signalen
  - synchrone Signale: durch Prozessaktivität ausgelöst (Analogie: Traps)
  - asynchrone Signale: "von außen" ausgelöst (Analogie: Interrupts)
- Zwecke von Signalen
  - Ereignissignalisierung zwischen Prozessen
  - Ereignissignalisierung des Betriebssystemkerns an einen Prozess

1 Reaktion auf Signale

- abort
  - erzeugt Core-Dump (Segmente + Registerkontext) und beendet Prozess
- exit
  - beendet Prozess, ohne einen Core-Dump zu erzeugen
- ignore
  - ignoriert Signal
- stop
  - stoppt Prozess
- continue
  - setzt einen gestoppten Prozess fort
- signal handler
  - Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

2 POSIX-Signalbehandlung

- Signal bewirkt Aufruf einer Funktion



  - nach der Behandlung läuft Prozess an unterbrochener Stelle weiter
- Systemschnittstelle
  - sigaction
  - sigprocmask
  - sigsuspend
  - kill

3 Signalhandler installieren: sigaction

- Prototyp

```
#include <signal.h>

int sigaction(int sig, /* Signal */
              const struct sigaction *act, /* Handler */
              struct sigaction *oact /* Alter Handler */ );
```

- Handler bleibt solange installiert, bis neuer Handler mit **`sigaction`** installiert wird
- sigaction-Struktur

```
struct sigaction {
    void (*sa_handler)(int); /* Behandlungsfunktion */
    sigset_t sa_mask; /* Signalmaske während der Behandlung */
    int sa_flags; /* diverse Einstellungen, bei uns 0 */
}
```


3 Signalhandler installieren: sigaction Handler (sa_handler)

- Signalbehandlung kann über **`sa_handler`** eingestellt werden:
  - ➤ **`SIG_IGN`** Signal ignorieren
  - ➤ **`SIG_DFL`** Default-Signalbehandlung einstellen
  - ➤ *Funktionsadresse* Funktion wird in der Signalbehandlung aufgerufen und ausgeführt
    - ➠ **`SIG_IGN`** und **`SIG_DFL`** wird werden über **exec(3)** vererbt, nicht aber eine Behandlungsfunktion (nicht möglich, warum?)
- Mit **`sa_flags`** lässt sich das Verhalten beim Signalempfang beeinflussen
  - ➤ bei uns gilt: **`sa_flags=0`**

3 Signalhandler installieren: sigaction Maske (sa_mask)

- verzögerte Signale
  - ◆ während der Ausführung der Signalhandler-Prozedur wird das auslösende Signal blockiert
  - ◆ bei Verlassen der Signalbehandlungsroutine wird das Signal deblockiert
  - ◆ es wird maximal ein Signal zwischengespeichert
- mit **`sa_mask`** in der **`struct sigaction`** kann man zusätzliche Signale während der Behandlung des Signals blockieren
- Auslesen und Modifikation von Signal-Masken des Typs **`sigset_t`** mit:
  - ◆ **`sigaddset()`**: Signal zur Maske hinzufügen
  - ◆ **`sigdelset()`**: Signal aus Maske entfernen
  - ◆ **`sigemptyset()`**: Alle Signale aus Maske entfernen
  - ◆ **`sigfillset()`**: Alle Signale in Maske aufnehmen
  - ◆ **`sigismember()`**: Abfrage, ob Signal in Maske enthalten ist

3 Signalhandler installieren: Beispiel

- Beispiel:

```
#include <signal.h>
void my_handler(int sig) { ... }
...
struct sigaction action;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
action.sa_handler = my_handler;
sigaction(SIGUSR1, &action, NULL); /* Fehlerbehandlung! */
```

4 Signal zustellen

- Systemaufruf **kill(2)**

```
int kill(pid_t pid, int signo);
```

- Kommando **kill(1)** aus der Shell (z. B. **`kill -USR1 <pid>`**)

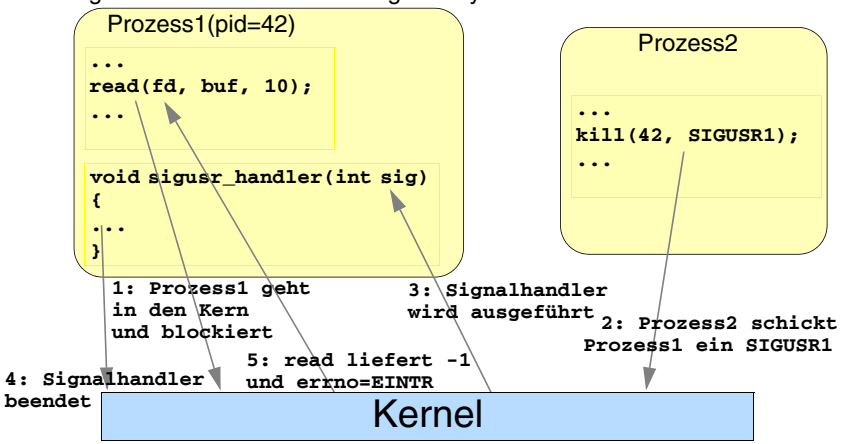
5 Ausgewählte POSIX Signale

Das Defaultverhalten bei den meisten Signalen ist die Terminierung des Prozesses, bei einigen Signalen mit Anlegen eines Core-Dumps.

- SIGALRM: Timer abgelaufen (**`alarm(2), setitimer(2)`**)
- SIGCHLD (ignore): Statusänderung eines Kindprozesses
- SIGINT: Interrupt; (Shell: CTRL-C)
- SIGQUIT (core): Quit; (Shell: CTRL-\)
- SIGKILL (nicht behandelbar): beendet den Prozess
- SIGTERM: Terminierung; Standardsignal für **`kill(1)`**
- SIGSEGV (core): Speicherschutzverletzung
- SIGUSR1, SIGUSR2: Benutzerdefinierte Signale

6 Unterbrechen von Systemaufrufen

- Signale können die Ausführung von Systemaufrufen unterbrechen



6 Unterbrechen von Systemaufrufen

- betrifft nur blockierende Systemcalls (z.B. **`wait()`**, **`read()`**)
- der Systemcall setzt dann **`errno=EINTR`** und kehrt mit Fehler zurück
- das Programm muss dies erkennen und
  - ➤ von einem echten Fehler unterscheiden ➠ Inspektion von **`errno`**
  - ➤ den Systemaufruf erneut starten

```
do {
    errno=0; /* evtl. früheren errno Wert überschreiben */
    ret=wait(NULL);
} while(ret == -1 && errno == EINTR);
if(errno != 0) { /* Fehlerbehandlung */ }
```

- Achtung: Dies betrifft auch Bibliotheksfunktionen, die diese Systemaufrufe verwenden
  - ➤ **fgets(3)**, **getchar(3)**, etc. verwenden **read(2)**
  - ➤ in der Manpage sollte dokumentiert sein, ob eine Funktion mit Fehlernummer **`errno=EINTR`** zurückkehren kann

7 Ändern der prozessweiten Signal-Maske

```
int sigprocmask(int how, /* Verknüpfung der Masken */
                const sigset_t *set, /* neue Maske */
                sigset_t *oset /* Speicher für alte Maske */ );
```

- how:
  - ◆ **`SIG_BLOCK`**: blockiert Signale der Maske (zusätzlich zu bereits blockierten)
  - ◆ **`SIG_SETMASK`**: blockiert Signale der Maske (und deblockiert alle anderen)
  - ◆ **`SIG_UNBLOCK`**: deblockiert Signale der Maske
- Beispiel

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
```

- Anwendung: Sperren der Signalbehandlung in kritischen Abschnitten
  Vgl.: Sperren der Interruptbehandlung in kritischen Abschnitten (**cli()**, **sei()**)
- **Achtung:** Die prozesseweite Signal-Maske wird über **exec(3)** vererbt!

8 Warten auf Signale

- Problem: Prozess will in einem kritischen Abschnitt auf ein Signal warten
  - ➤ Signal muss deblockiert werden
  - ➤ Prozess wartet auf Signal
  - ➤ Signal muss wieder blockiert werden
- Operationen müssen atomar am Stück ausgeführt werden!
  ➠ gleiche Problematik wie bei den Stromsparmodi des AVR-Prozessors
- Prototyp

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

  - ◆ **`sigsuspend(mask)`** setzt **`mask`** als Signal-Maske und blockiert Prozess
  - ◆ Signal führt zu Aufruf des Signalhandlers (muss vorher installiert werden)
  - ◆ **`sigsuspend`** restauriert die ursprüngliche Signal-Maske und kehrt zurück

9 POSIX-Signale vs. Interrupts

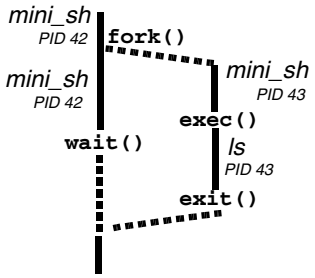
	Signale	Interrupts
Behandlung installieren	sigaction()	`ISR()`-Makro der C-Bibliothek
Behandlungsfunktion	Signalhandler	Interrupthandler
Auslösung	durch Prozesse mit kill() oder durch das Betriebssystem	durch die Hardware
Synchronisation	sigprocmask()	cli(), sei()
Warten auf IRQ/Signal	sigsuspend(), pause()	sleep_cpu()

- Signale und Interrupts sind sehr ähnliche Konzepte auf unterschiedlichen Ebenen
- Viele Probleme treten in beiden Fällen auf und sind konzeptionell identisch zu lösen

U7-5 Aufgabe 6: 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

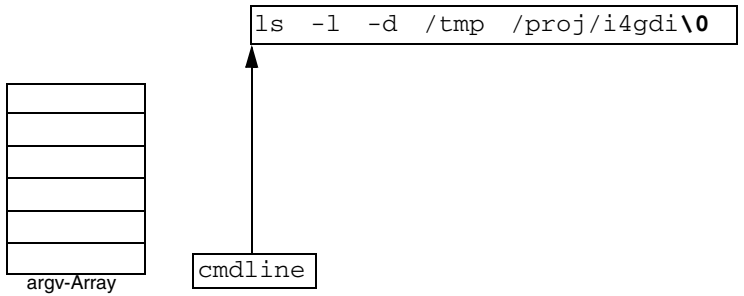
2 strtok

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

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

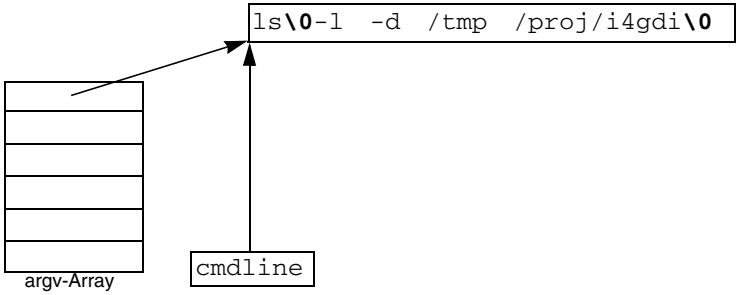
- Wird sukzessive aufgerufen und liefert jeweils einen Zeiger auf das nächste Token (mehrere aufeinanderfolgende Trennzeichen werden hierbei übersprungen)
  - `str` ist im ersten Aufruf ein Zeiger auf den zu teilenden String, in allen Folgeaufrufen `NULL`
  - `delim` ist ein String, der alle Trennzeichen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

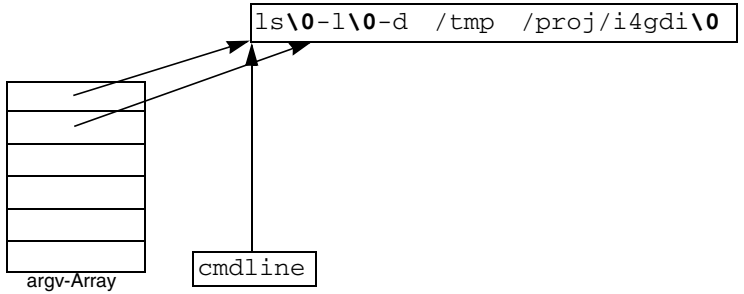


- Erster **strtok**-Aufruf mit dem Zeiger auf diesen Speicherbereich

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

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

2 strtok-Beispiel

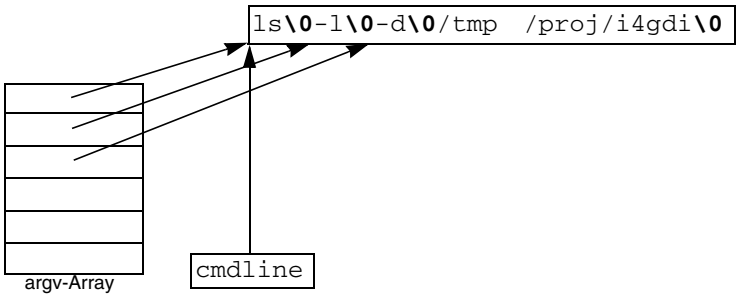


- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger

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

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

2 strtok-Beispiel

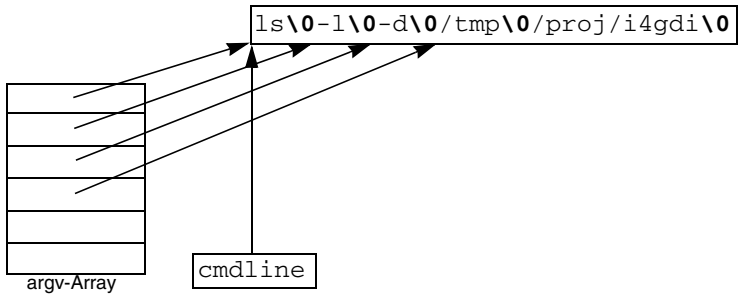


- Weitere Aufrufe von **strtok** nun mit einem NULL-Zeiger

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

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

2 strtok-Beispiel

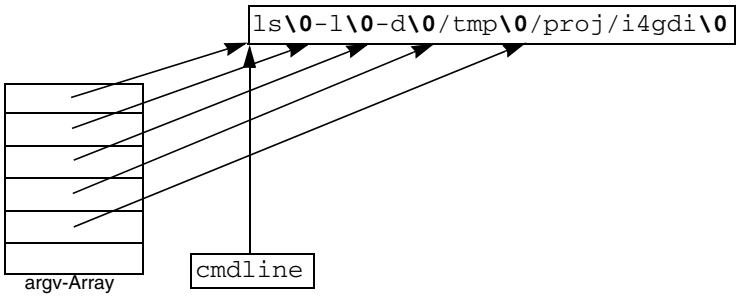


- Weitere Aufrufe von **strtok** nun mit einem NULL-Zeiger

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

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

2 strtok-Beispiel

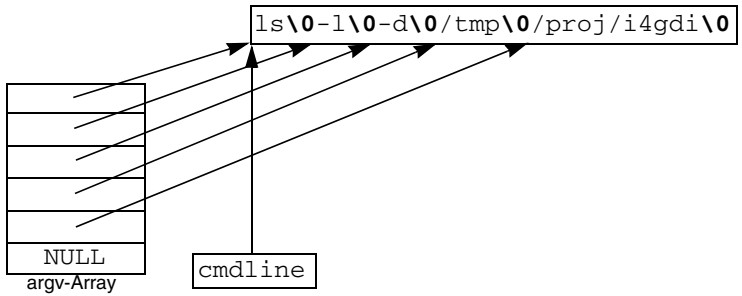


- Weitere Aufrufe von **strtok** nun mit einem NULL-Zeiger

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

- **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");
}
```

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