Übungen zu Systemprogrammierung 1 (SP1)

Ü5 – Prozesse

Jens Schedel, Christoph Erhardt, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 2014 – 19. bis 23. Mai 2014

http://www4.cs.fau.de/Lehre/SS14/V_SP1

Agenda

5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

Agenda

5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

Aufteilung des Adressraums

- Aufteilung des Hauptspeichers eines Prozesses in Segmente
  - Vgl. Vorlesung *A/V, Seite 7f.*

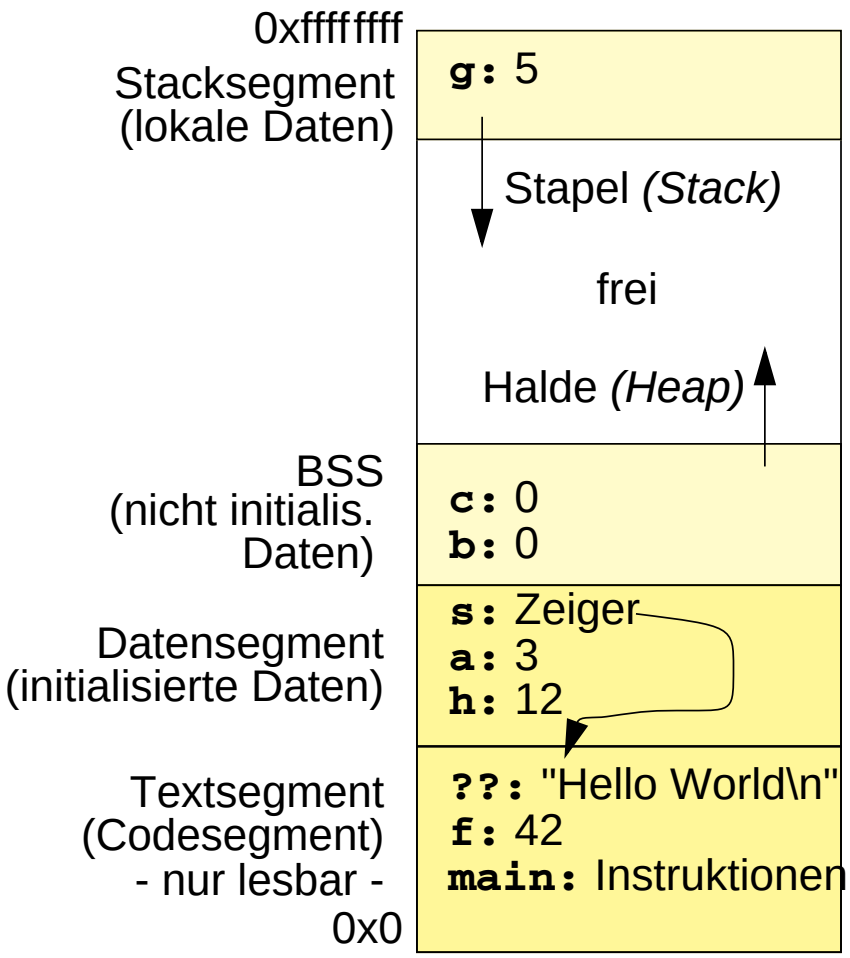
```
static int a = 3; static int b;
static int c = 0; const int f = 42;
const char *s = "Hello World\n";

int main(void) {
    int g = 5;
    static int h = 12;
}
```

- Compiler-Fehler

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

- Segmentation Fault

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



Agenda

5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

Exkurs: Vernünftige Speicherverwaltung

- Je nach Segment haben Daten unterschiedliche Lebensdauer
  - Stack (lokal nicht-`static`): bis Verlassen des umgebenden Blocks
  - Daten (global / lokal `static`): „unsterblich“ – bis zum Prozessende
  - Heap (dynamisch alloziert mit `malloc(3)`):
    - Bis zur expliziten Freigabe mit `free(3)`
    - Nachträgliche Größenänderung mit `realloc(3)` möglich
- `malloc(3)` ist am flexibelsten – aber nicht immer die beste Lösung!
  - Allokation kostet Zeit
  - Aufwändiger Code, Fehlerbehandlung nötig
  - Freigabe darf nicht vergessen werden
- Oft die bessere Wahl: lokales Array auf dem Stack
  - Voraussetzung 1: beschränkte Lebensdauer okay
  - Voraussetzung 2: keine nachträgliche Größenänderung
    - d. h. obere Schranke für Größe muss vorab ermittelbar sein

Agenda

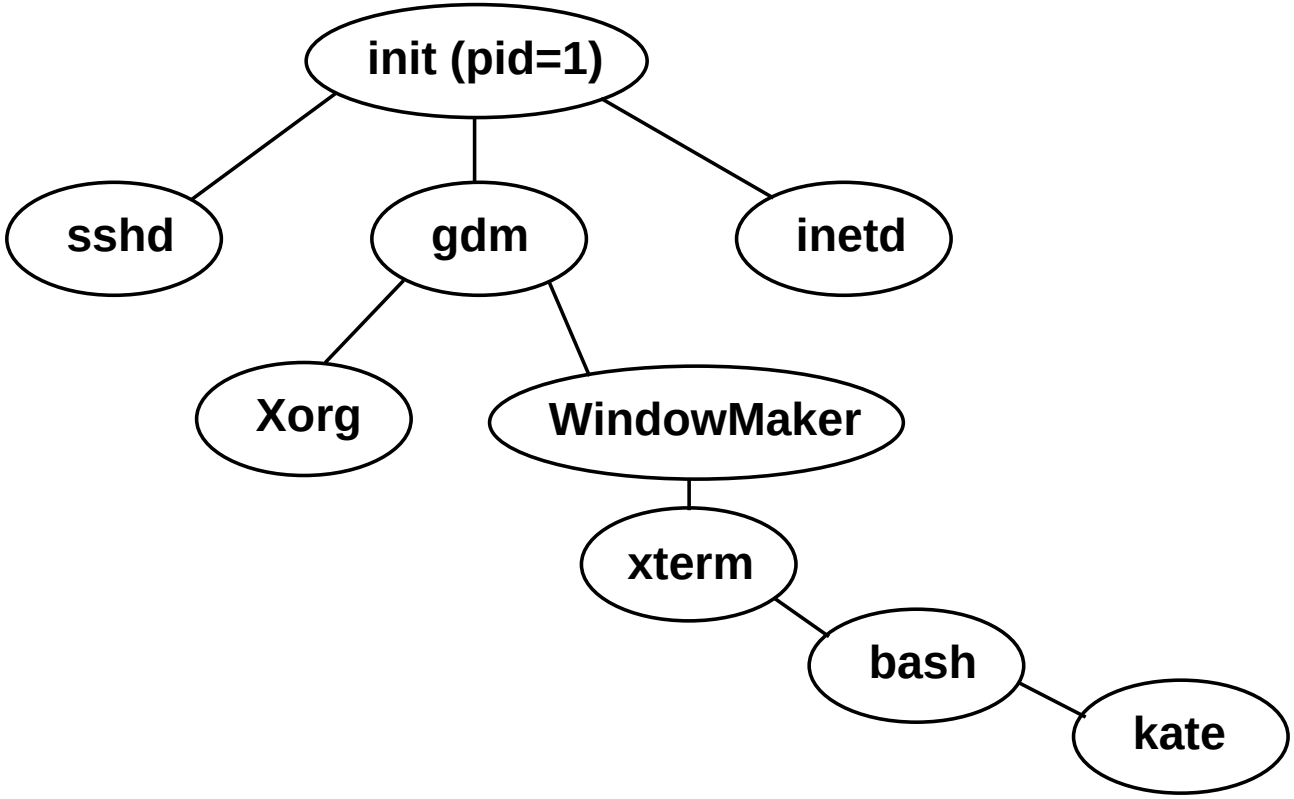
5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

Prozesse

- 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

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**

Agenda

5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

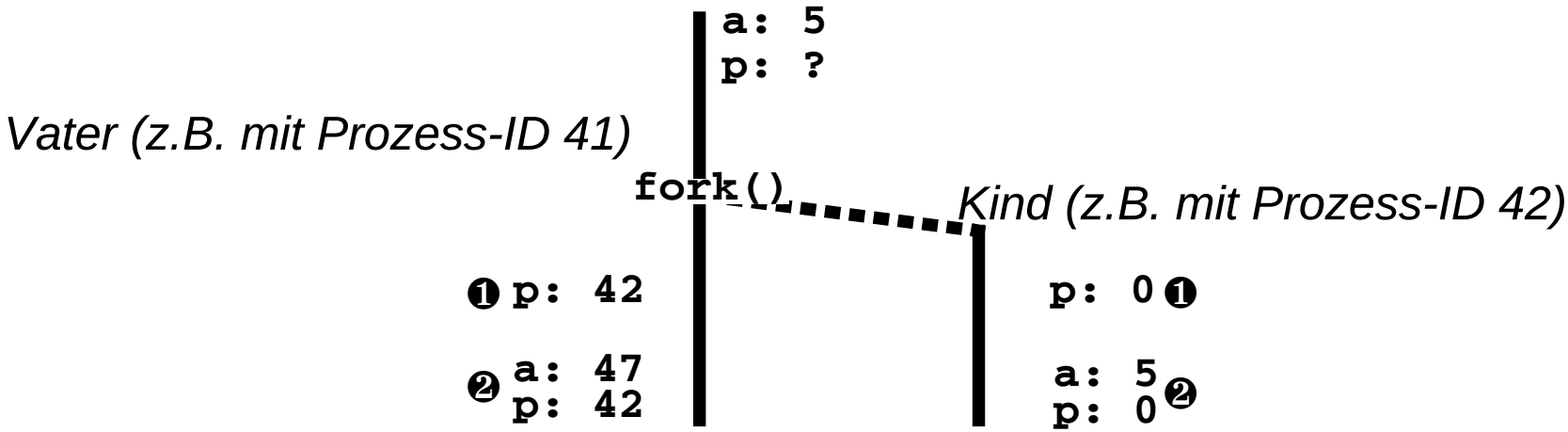
Kindprozess erzeugen – fork(2)

```
pid_t fork(void);
```

- 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)
  - Dateideskriptoren (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

Kindprozess erzeugen – fork(2)

```
int a = 5;
pid_t p = fork(); // (1)
a += p; // (2)
if ( p == -1 ) {
  // fork-Fehler, es wurde kein Kind erzeugt
  ...
} else if ( p == 0 ) {
  // Hier befinden wir uns im Kind
  ...
} else {
  // Hier befinden wir uns im Vater
  // p ist die PID des neu erzeugten Kindprozesses
  ...
}
```



Programm ausführen – exec(3)

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- Lädt Programm zur Ausführung in den aktuellen Prozess (vgl. Vorlesung A | III-5.3)
  - aktuell ausgeführtes Programm wird ersetzt (Text-, Daten- und Stacksegment)
  - erhalten bleiben: Dateideskriptoren (= geöffnete Dateien), Arbeitsverzeichnis, ...
- Aufrufparameter für `exec(3)`
  - Dateiname des neuen Programmes
  - Argumente, die der `main`-Funktion des neuen Programms übergeben werden
- `exec` kehrt nur im Fehlerfall zurück

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[]);
```

- Anmerkung: Alle Varianten von `exec(3)` erwarten als letzten Eintrag in der Argumentenliste einen `NULL`-Zeiger.

Prozess beenden – exit(3)

```
void exit(int status);
```

- beendet aktuellen Prozess mit angegebenem Exitstatus
- gibt alle Ressourcen frei, die der Prozess belegt hat, z. B.
  - Speicher
  - Dateideskriptoren (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 (mit `wait(2)` bzw. `waitpid(2)`)!
  - 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

Auf Kindprozess warten – wait(2)

```
pid_t wait(int *status);
```

- `wait(2)` liefert Informationen über einen terminierten Kindprozess (*Zombie*):
  - PID dieses Kindprozesses wird als Rückgabewert geliefert
  - als Parameter kann ein Zeiger auf einen `int`-Wert mitgegeben werden, in dem unter anderem der Exitstatus des Kindprozesses abgelegt wird
  - in den Status-Bits wird eingetragen, „was dem Kindprozess zugestoßen ist", Details können über Makros abgefragt werden:
    - Prozess mit `exit(3)` terminiert: `WIFEXITED(status)`
    - Exitstatus: `WEXITSTATUS(status)` = Argument, das an `exit(3)` übergeben wurde
    - weitere siehe `wait(2)`
- Verbleibende Ressourcen des Zombies werden aufgeräumt
- Falls aktuell kein Kindprozess im Zombie-Zustand ist, wartet `wait(2)` bis zum Terminieren des nächsten Kindprozesses und räumt diesen dann ab

Auf speziellen Kindprozess warten – waitpid(2)

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

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

Agenda

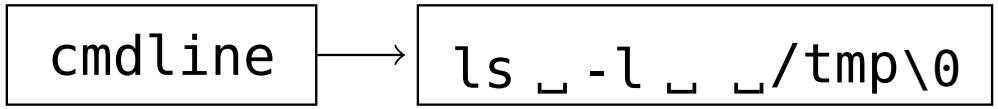
5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

String-Manipulation mit `strtok(3)`

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

- `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

String-Manipulation mit `strtok(3)`



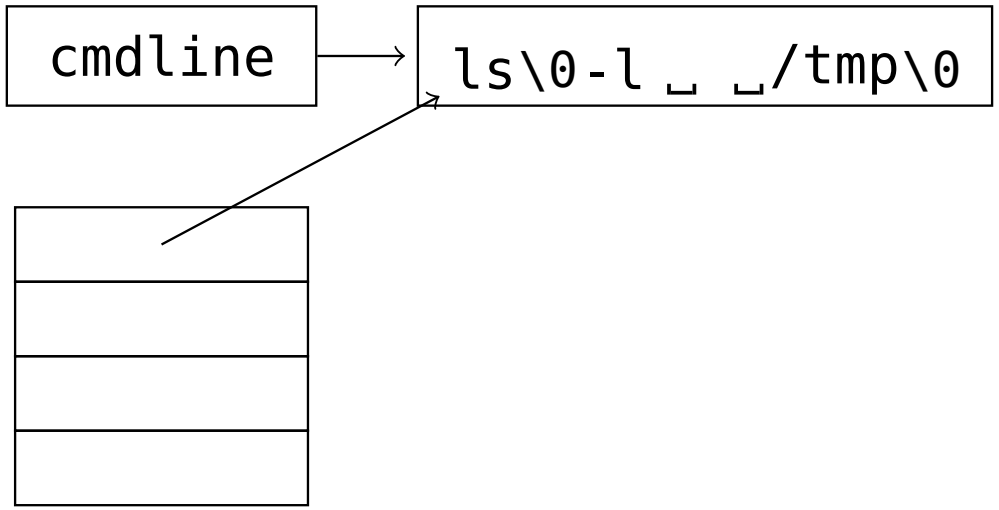

```
a[0] = strtok(cmdline, " ");
```

```
a[1] = strtok(NULL, " ");
a[2] = strtok(NULL, " ");
```

```
a[3] = strtok(NULL, " ");
```

- Kommandozeile liegt als `'\0'`-terminierter String im Speicher

String-Manipulation mit `strtok(3)`



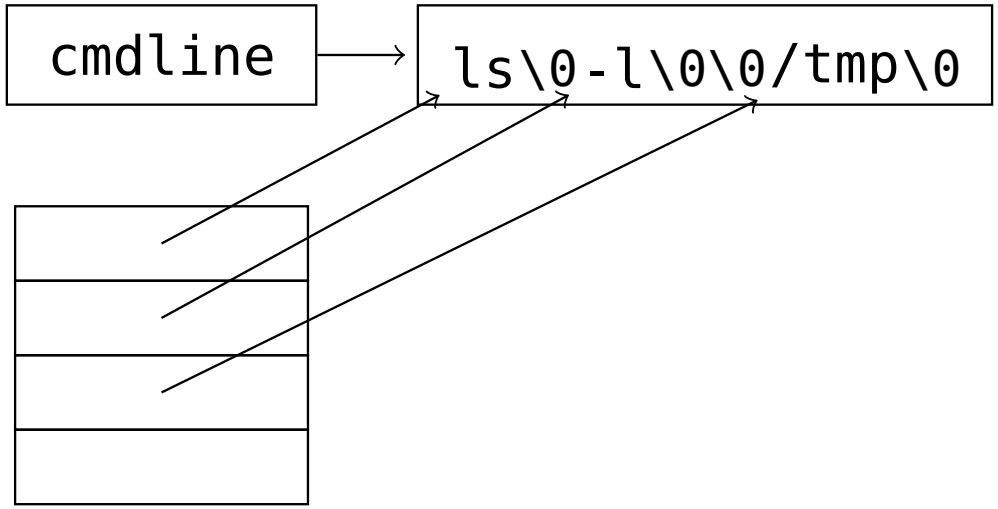

```
a[0] = strtok(cmdline, " ");
```

```
a[1] = strtok(NULL, " ");
a[2] = strtok(NULL, " ");
```

```
a[3] = strtok(NULL, " ");
```

- Kommandozeile liegt als `'\0'`-terminierter String im Speicher
- Erster `strtok(3)`-Aufruf mit dem Zeiger auf diesen Speicherbereich liefert Zeiger auf erstes Token `ls` und ersetzt den Folgetrenner mit `'\0'`

String-Manipulation mit `strtok(3)`



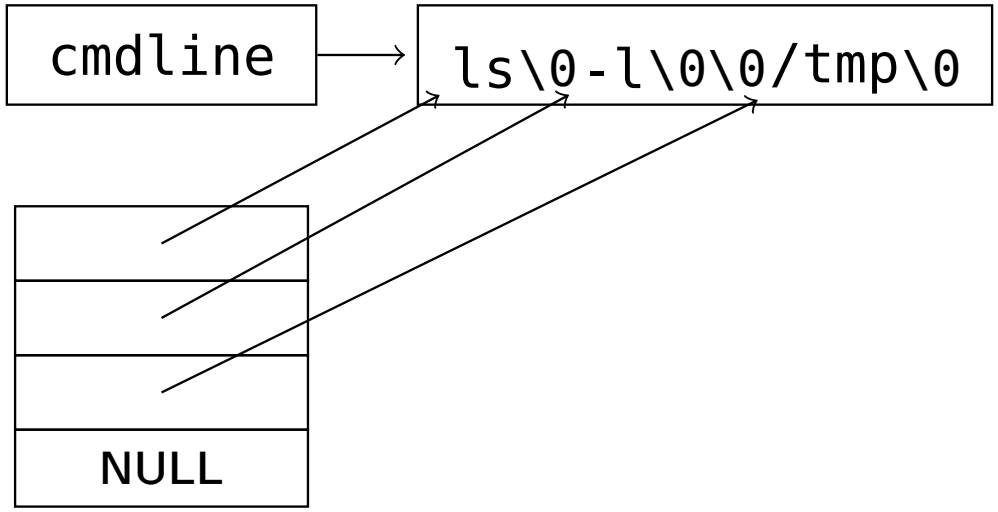

```
a[0] = strtok(cmdline, " ");
```

```
a[1] = strtok(NULL, " ");
a[2] = strtok(NULL, " ");
```

```
a[3] = strtok(NULL, " ");
```

- Kommandozeile liegt als `'\0'`-terminierter String im Speicher
- Erster `strtok(3)`-Aufruf mit dem Zeiger auf diesen Speicherbereich liefert Zeiger auf erstes Token `ls` und ersetzt den Folgetrenner mit `'\0'`
- Weitere Aufrufe von `strtok(3)` nun mit einem `NULL`-Zeiger liefern jeweils Zeiger auf das nächste Token

String-Manipulation mit `strtok(3)`




```
a[0] = strtok(cmdline, " ");
```

```
a[1] = strtok(NULL, " ");
a[2] = strtok(NULL, " ");
```

```
a[3] = strtok(NULL, " ");
```

- Kommandozeile liegt als `'\0'`-terminierter String im Speicher
- Erster `strtok(3)`-Aufruf mit dem Zeiger auf diesen Speicherbereich liefert Zeiger auf erstes Token `ls` und ersetzt den Folgetrenner mit `'\0'`
- Weitere Aufrufe von `strtok(3)` nun mit einem `NULL`-Zeiger liefern jeweils Zeiger auf das nächste Token
- Am Ende liefert `strtok(3)` `NULL`

Agenda

5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

Pseudo-Targets

- Dienen nicht der Erzeugung einer gleichnamigen Datei
  - so deklarierte Targets werden immer gebaut
  - Deklaration als Abhängigkeit des Spezial-Targets `.PHONY` nötig
- Beispiel: Installation einer ausführbaren Datei mit `make install`

```
.PHONY: install

install: clash
  cp clash /usr/bin
```

  - Konventionen
    - `all` ist immer erstes Target im Makefile und baut die komplette Anwendung
    - `clean` löscht alle durch `make` erzeugte Dateien

Agenda

5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

Debugger: gdb

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man u.a.
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren
    - Erlauben von core dumps (in der laufenden Shell): z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`
- Programm sollte Debug-Symbole enthalten
- Aufruf des Debuggers mit `gdb <Programmname>`

Beispiel

```
void initArray(long *array, unsigned int size) {
  int i;
  for ( i=0; i<=size; i++ ) {
    array[i] = 0;
  }
}

int main(int argc, char *argv[]) {
  long *array;
  long buf[7];
  array = buf;

  initArray(buf,sizeof(buf)/sizeof(long));

  while ( array != buf+sizeof(buf)/sizeof(long) ) {
    printf("%ld\n", *array);
    array++;
  }

  exit(EXIT_SUCCESS);
}
```


Befehlsübersicht

- Programmausführung beeinflussen
  - Breakpoints setzen:
    - `b [<Dateiname>:]<Funktionsname>`
    - `b <Dateiname>:<Zeilennummer>`
  - Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
  - Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
  - schrittweise Abarbeitung auf Ebene der Quellsprache mit
    - `s` (step: läuft in Funktionen hinein)
    - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)
  - Breakpoints anzeigen: `info breakpoints`
  - Breakpoint löschen: `delete breakpoint#`

Befehlsübersicht

- Variableninhalte anzeigen/modifizieren
  - Anzeigen von Variablen mit: `p expr`
    - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
  - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
  - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`
- Quellcode an aktueller Position anzeigen: `list`
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
  - `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
  - `rwatch expr`: Stoppt, wenn `expr` gelesen wird
  - `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
  - Anzeigen und Löschen analog zu den Breakpoints

Agenda

5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

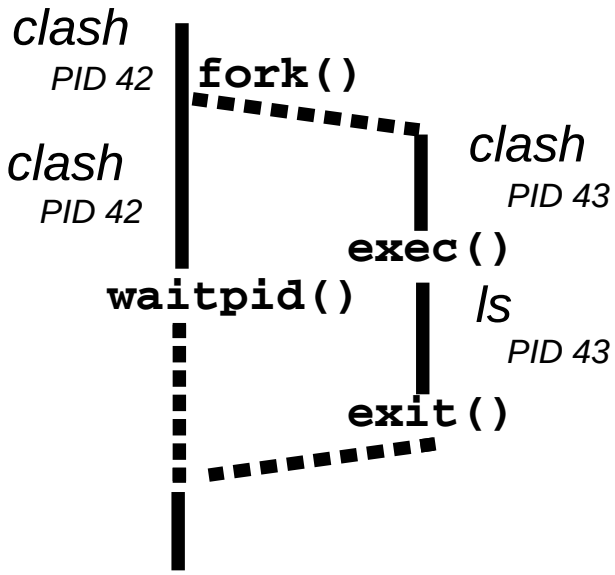
Ziele der Aufgabe

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

Funktionsweise der clash

- Eingabezeile, aus der der Benutzer Programme starten kann
  - Länge der Eingabezeile und damit Anzahl der Argumente unbekannt
  - Abfrage von Konfigurationsoptionen des Betriebssystems mit `sysconf(3)`
    - Maximale Länge einer Zeichenkette, die auf einmal eingelesen werden kann (`stdin` oder Datei) (`_SC_LINE_MAX`)

- 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. Exitstatus wird bei der Anzeige des Promptes ausgegeben



Agenda

5.1 Adressraumstruktur
5.2 Exkurs: Vernünftige Speicherverwaltung
5.3 Prozesse
5.4 System-Schnittstelle
5.5 String-Manipulation mit `strtok(3)`
5.6 Make
5.7 gdb
5.8 Aufgabe 4: clash
5.9 Gelerntes anwenden

Aktive Mitarbeit!

„Aufgabenstellung“

- Programm schreiben, welches ein Kommando mit jedem der übergebenen Parameter einmal ausführt.
  - `./listRun <program> <arguments...>`
  - Beispiel:

```
$ ./listRun echo Das ist ein Test
Das
ist
ein
Test
```

- Optional: `arguments`-Array vor dem Ausführen sortieren