

Übungen zu Systemprogrammierung 1

Ü4 – Prozesse

Sommersemester 2020

Dustin Nguyen, Jonas Rabenstein, Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

5.1 Adressraumstruktur

5.2 Prozesse

5.3 System-Schnittstelle

5.4 String-Manipulation mit strtok(3)

5.5 Make

5.6 Aufgabe 4: clash

5.7 Gelerntes anwenden



5.1 Adressraumstruktur

5.2 Prozesse

5.3 System-Schnittstelle

5.4 String-Manipulation mit strtok(3)

5.5 Make

5.6 Aufgabe 4: clash

5.7 Gelerntes anwenden

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

- Vgl. Vorlesung A-III, 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

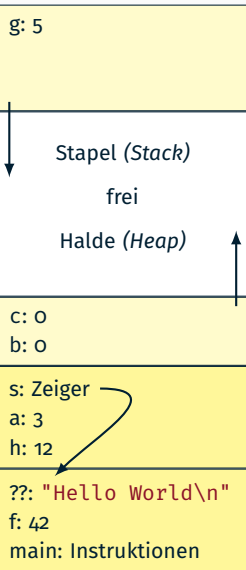
```
s[1] = 'a';  
f = 2;
```

■ Segmentation Fault

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

0xffff ffff

Stacksegment
(lokale Daten)



Textsegment
(Code segment)
nur lesbar
0x0



- 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
 - Voraussetzung 3: Array passt ins Stack-Segment (übliche Stack-Größe z. B. 8 MiB, `ulimit -a`)



5.1 Adressraumstruktur

5.2 Prozesse

5.3 System-Schnittstelle

5.4 String-Manipulation mit strtok(3)

5.5 Make

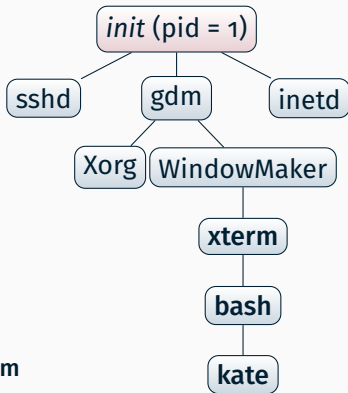
5.6 Aufgabe 4: clash

5.7 Gelerntes anwenden



- Prozesse sind eine Ausführungsumgebung für Programme (vgl. Vorlesung A-III, S. 7)
 - haben eine Prozess-ID (PID, ganzzahlig positiv)
 - führen ein Programm aus
- Mit einem Prozess sind Ressourcen verknüpft, z. B.
 - Speicher
 - Adressraum
 - offene Dateien

- Zwischen Prozessen bestehen Elter-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**





5.1 Adressraumstruktur

5.2 Prozesse

5.3 System-Schnittstelle

5.4 String-Manipulation mit strtok(3)

5.5 Make

5.6 Aufgabe 4: clash

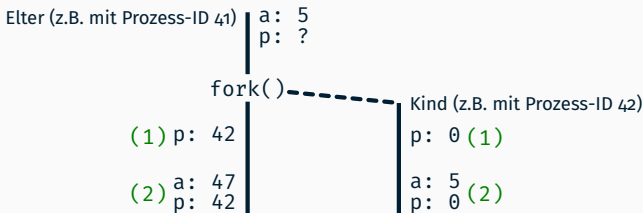
5.7 Gelerntes anwenden



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

- Erzeugt einen neuen Kindprozess (Vorlesung A-III, S. 14ff.)
- (Fast) exakte Kopie des Elterprozesses ...
 - Datensegment (neue Kopie, gleiche Daten)
 - Stacksegment (neue Kopie, gleiche Daten)
 - Textsegment (gemeinsam genutzt, da nur lesbar)
 - Dateideskriptoren (geöffnete Dateien)
 - Registerinhalte
 - ... mit Ausnahme der Prozess-ID
- Kind startet Ausführung hinter dem fork(2) mit dem geerbten Zustand
 - das ausgeführte Programm muss anhand der PID (Rückgabewert von fork(2)) entscheiden, ob es sich um den Elter- oder den Kindprozess handelt

```
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 Elterprozess
    // p ist die PID des neu erzeugten Kindprozesses
    ...
}
```





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

- Lädt Programm zur Ausführung in den aktuellen Prozess (vgl. Vorlesung A-III, S. 10f.)
 - 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



- Varianten:

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

// Argumente als Array
int execvp(const char *file, char *const argv[]);
```

- Zum Suchen von file wird die Umgebungsvariable PATH verwendet
- Name der Programmdatei muss **doppelt** übergeben werden – als file und als arg0 bzw. argv[0]!
- Alle Varianten von exec(3) erwarten als letzten Eintrag in der Argumentenliste bzw. im argv-Array einen **NULL**-Zeiger



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

- beendet aktuellen Prozess mit angegebenem Exitstatus
 - `EXIT_SUCCESS` = 0
 - `EXIT_FAILURE` != 0, meist 1
- gibt alle Ressourcen frei, die der Prozess allokiert hat, z. B.
 - Speicher
 - Dateideskriptoren (schließt alle offenen Dateien)
- Prozess geht in den *Zombie*-Zustand über
 - ermöglicht es dem Elterprozess auf den Tod des Kindes zu reagieren
 - Zombie-Prozesse belegen Ressourcen und sollten zeitnah beseitigt werden (mit `wait(2)` bzw. `waitpid(2)`)!
 - ist der Elterprozess schon vor dem Kind terminiert, so wird der Zombie an den Prozess mit PID 1 (z. B. *init*) weitergereicht, welcher diesen sofort beseitigt



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

- 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(wstatus)`
 - Exitstatus: `WEXITSTATUS(wstatus)` = Argument, das an `exit(3)` übergeben wurde
 - weitere siehe `wait(2)`
- Verbleibende Ressourcen des Zombies werden aufgeräumt
 - PID wird als frei markiert
 - Prozesskontrollblock wird freigegeben
- Falls aktuell kein Kindprozess im Zombie-Zustand ist, wartet `wait(2)` bis zum Terminieren des nächsten Kindprozesses und räumt diesen dann ab



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

- Mächtigere Variante von `wait(2)`
- Wartet auf Statusänderung eines
 - bestimmten Prozesses: `pid > 0`
 - beliebigen Kindprozesses: `pid == -1`
 - weitere siehe `waitpid(2)`
- 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



5.1 Adressraumstruktur

5.2 Prozesse

5.3 System-Schnittstelle

5.4 String-Manipulation mit strtok(3)

5.5 Make

5.6 Aufgabe 4: clash

5.7 Gelerntes anwenden



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

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

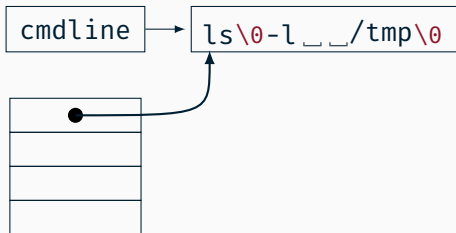
cmdline → ls -l /tmp\0

```
cmdline = "ls -l /tmp";  
a[0] = strtok(cmdline, " ");  
a[1] = strtok(NULL, " ");  
a[2] = strtok(NULL, " ");  
a[3] = strtok(NULL, " ");
```

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

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

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

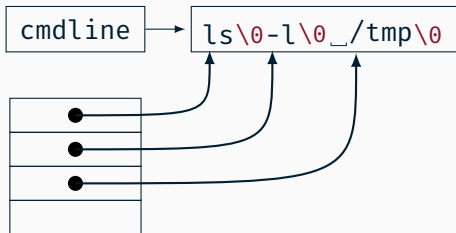


```
cmdline = "ls -l /tmp";  
a[0] = strtok(cmdline, " ");  
a[1] = strtok(NULL, " ");  
a[2] = strtok(NULL, " ");  
a[3] = strtok(NULL, " ");
```

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

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

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

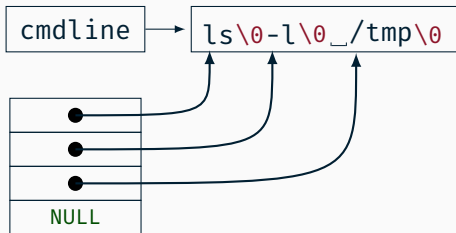


```
cmdline = "ls -l /tmp";  
a[0] = strtok(cmdline, " ");  
a[1] = strtok(NULL, " ");  
a[2] = strtok(NULL, " ");  
a[3] = strtok(NULL, " ");
```

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

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

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



```
cmdline = "ls -l /tmp";  
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 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`



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

- `delim` ist ein String, der alle Trennzeichen enthält, z. B. " \t\n"
- `strtok(3)` nutzt jedes *Zeichen* in `delim` als mögliches Trennzeichen
- Mehrere aufeinanderfolgende Trennzeichen werden übersprungen
- Vorsicht: `strtok(3)` gibt bei Strings nur aus Trennzeichen `NULL` zurück, ohne den String zu verändern!



5.1 Adressraumstruktur

5.2 Prozesse

5.3 System-Schnittstelle

5.4 String-Manipulation mit strtok(3)

5.5 Make

5.6 Aufgabe 4: clash

5.7 Gelerntes anwenden



- Dienen nicht der Erzeugung einer gleichnamigen Datei
 - so deklarierte Targets werden immer gebaut
 - Deklaration als Abhängigkeit des Spezial-Targets **.PHONY** nötig

- Beispiel: Erzeugen einer ausführbaren Datei mit `make all`

```
.PHONY: all clean
```

```
all: clash
```

```
clean:  
    rm -f clash
```

```
clash: clash.o #...  
    # build clash
```

- Konventionen
 - `all` ist immer erstes Target im Makefile und baut die komplette Anwendung
 - `clean` löscht alle durch `make` erzeugte Dateien
 - **Hinweis:** bei Aufruf von `rm` den Parameter `-f` verwenden
⇒ kein Abbruch bei nicht existierenden Dateien



5.1 Adressraumstruktur

5.2 Prozesse

5.3 System-Schnittstelle

5.4 String-Manipulation mit strtok(3)

5.5 Make

5.6 Aufgabe 4: clash

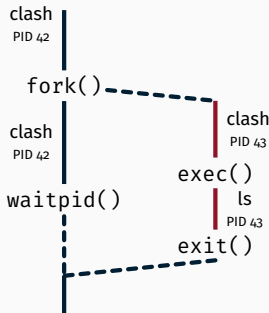
5.7 Gelerntes anwenden



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

- (Verbindliche) Schnittstellendokumentation zu `plist.c` auf der Website (Übung -> Aufgaben)

- Eingabezeile, aus der der Benutzer Programme starten kann
 - Länge der Eingabezeile und damit Anzahl der Argumente unbekannt
 - Für die clash (willkürlich) auf 1337 Zeichen beschränkt
- 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





5.1 Adressraumstruktur

5.2 Prozesse

5.3 System-Schnittstelle

5.4 String-Manipulation mit strtok(3)

5.5 Make

5.6 Aufgabe 4: clash

5.7 Gelerntes anwenden

„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