Aufgabe 3: Prozesse

In den folgenden Aufgabe soll der grundlegende Umgang mit dem UNIX-Prozessmodell, das Sie in der Vorlesung kennengelernt haben, geübt werden. Machen Sie sich als Vorbereitung mit den folgenden Man-Pages vertraut: **fork(3)**, **waitpid(2)**, **exec(3)**, **exit(3)**. Suchen Sie dabei gezielt nach folgenden Informationen:

- Was tut die Funktion im Groben?
- Welche Parameter müssen Sie übergeben und was enthalten diese Parameter?
- Was ist der Rückgabewert?
- Wenn es Optionen gibt, was bewirken diese?

Im Laufe der folgenden Aufgaben werden Sie eine eigene Shell implementieren. Die Aufgaben bauen aufeinander auf und sollten (wenn nicht anders angegeben) in der gegebenen Reihenfolge bearbeitet werden. Scheuen Sie bei der Bearbeitung nicht davor, eigene Hilfsfunktionen zu erstellen.

Achtung: Die Bewertung dieser Aufgabe erfolgt mit Hilfe von Tests, welche Ihnen bereitgestellt werden. **Tests, die nicht ordnungsgemäß funktionieren, geben keine Punkte.**

3.1 Eingabeaufforderung (2 Punkte)

Öffnen Sie das Projekt, wie zuvor in der Aufgabe 2R, mittels Qt-Creator. Schauen Sie sich zuerst die Datei `main.c` an und machen Sie sich groben Ablauf des Programms vertraut. Als erstes sollte Ihre Shell in der Lage sein, Befehle entgegen zu nehmen. Implementieren Sie hierfür die Funktion `read_input()` in der Datei `shellfunctions.c`. Diese soll eine Eingabeaufforderung mittels `prompt()` aus dem *shellutils*-Modul ausgeben und anschließend auf die Eingabe einer Zeile warten (s. **fgets(3)**).

Es soll hier außerdem getestet werden, ob die Shell beendet werden soll. Besteht die Eingabe nur aus einem `End-of-File (EOF)`-Zeichen (entspricht `Strg+d`), dann soll in der nächsten Zeile ***exakt*** die Meldung `Goodbye!` ausgegeben und die *myshell* mittels **exit(3)** beendet werden. Prüfen Sie hierfür, ob **fgets(3)** als Rückgabewert `NULL` zurückgibt, und prüfen Sie anschließend mit **feof(3)**, ob auf dem Standardeingabekanal (`stdin`) ein `EOF` liegt.

Anmerkung: Testen Sie Ihre Implementierung in einem Terminal und nicht direkt in Qt und mit dem Test `test-input`.

3.2 Ausführen von Vordergrundprozessen (9 Punkte)

Erweitern Sie nun die *myshell* um die Ausführung von Programmen. Implementieren Sie dazu die Funktion `execute_command()` in der `shellfunctions.c`. Diese Funktion sollte zunächst die gelesene Zeile in ein bequemes Datenformat (s. `command_t` in `shellutils.h`) überführen. Verwenden Sie hierfür die Funktion `parse_command_line()` aus dem *shellutils*-Modul. Beachten Sie dabei, dass diese Funktion Speicher dynamisch alloziert und ***Sie*** diesen wieder freigeben müssen.

Fangen Sie Fehler ab, wie zurückgegebene `NULL`-Zeiger, den *parse_error* (s. `command_t`) sowie den Speziallfall eines leeren Befehls. Orientieren Sie sich im letzteren Fall am Verhalten der **bash(1)**.

Wurde `parse_command_line()` richtig ausgeführt, erzeugen Sie einen neuen Kind-Prozess (**fork(3)**) und warten Sie dann im Elternprozess auf die Terminierung dieses Kind-Prozesses (**waitpid(2)**). Führen Sie das Programm mittels `execvp()` (s. **exec(3)**) im neu erzeugten Kind-Prozess aus. Geben Sie nach Beendigung des Kindprozesses mit der Funktion `print_status()` aus dem *shellutils*-Modul den Exit-Status aus.

Bibliotheksfunktionen wie **fork(3)**, **exec(3)** oder **waitpid(2)** können fehlschlagen, z.B. wenn bei `execvp()` das Programm nicht ausgeführt werden kann. Fangen Sie mögliche Fehler ab und geben Sie die Fehlermeldung mittels **perror(3)** aus.

Anmerkung: Testen Sie mit Hilfe verschiedener Programme (`sleep`, `ls`) und dem Test `test-foreground`, ob diese korrekt ausgeführt werden.

3.3 Ausführung von Hintergrundprozessen (2 Punkte)

Wenn die Ausführung von Vordergrundprozessen funktioniert, erweitern Sie die *myshell* um die Ausführung von Prozessen im Hintergrund. Erweitern Sie hierzu die Funktion `execute_command()` in der `shellfunctions.c` entsprechend.
Endet eine Eingabe mit dem Token „`&`“, d.h. in der Kommandostruktur ist `background==1`, so wird das ausgeführte Programm als Hintergrundprozess ausgeführt. In diesem Fall soll sich die *myshell* wie die `bash` verhalten und nicht auf die Terminierung des Prozesses warten, sondern sofort eine neue Eingabeaufforderung zeigen.

Testen Sie wieder mithilfe verschiedener, Ihnen bekannter Shell-Befehle, ob diese korrekt ausgeführt werden. Beispielsweise könnten `gvim&`, `gedit&` oder `nautilus&` ausgeführt werden. Hierbei sollte es anschließend noch möglich sein, weitere Befehle in der Shell auszuführen.

Anmerkung: Sie können dies mit dem Test `test-background` testen.

3.4 Behandlung von Zombies (2 Punkte)

Sie haben erfolgreich Hintergrundprozesse implementiert und können so mehrere Prozesse parallel ausführen. Allerdings können diese irgendwann unbemerkt terminieren, ohne dass ihr Exit-Status abgefangen wird. Dies sind sogenannte **Zombieprozesse**. Ihre Aufgabe ist es nun, das Problem der Zombies zu beheben.

- Prüfen Sie vor der Anzeige eines neuen Prompts, welche Hintergrundprozesse bereits terminiert sind.
- Fangen Sie den Exit-Status dieser Prozesse ab (s. **waitpid(2)**).
- Geben Sie analog zu den Vordergrundprozessen den Exit-Status aus. Beachten Sie bitte, dass die Kommandozeile, mit der ein Prozess gestartet wurde, in der verketteten Liste (s. *plist*) gespeichert werden soll und von der Funktion `remove_element()` über die Parameter zurückgegeben wird.

Implementieren Sie dazu die Funktion `collect_defunct_process()` in der `shellfunctions.c`.

Anmerkung: Zum Testen können Sie das Programm **sleep(1)** und den Test `test-zombies` nutzen.

3.5 Verzeichniswechsel (2 Punkte)

Diese Aufgabe kann auch direkt nach Aufgabe 3.1 bearbeitet werden.

Bisher kann die *myshell* Prozesse ausführen und sich mit Zombies herumschlagen. Allerdings geht das immer nur in dem gleichen, statischen Arbeitsverzeichnis, aus dem die *myshell* gestartet wurde. Das soll sich jetzt ändern.
Implementieren Sie in `execute_command()` ein internes Kommando „`cd`“ (cd für ***c**hange **d**irectory*).

Prüfen Sie hierfür mit **strcmp(3)**, ob in der Kommando-Eingabe das erste Wort „`cd`“ ist und prüfen Sie im Erfolgsfall, ob exakt ein Parameter übergeben wurde. Ist dies der Fall, führen Sie einen Wechsel des Arbeitsverzeichnisses mittels **chdir(3)** durch.

Bei einem Verzeichniswechsel kann es passieren, dass dieser nicht durchgeführt werden kann, weil z.B. das Verzeichnis nicht existiert. Sorgen Sie bitte in diesem Fall für eine angemessene Fehlerausgabe mittels **perror(3)**.

Anmerkung: Testen Sie Ihre Implementierung, indem Sie „`cd`“ und den Test `test-cd` nutzen.

3.6 Anzeige laufender Hintergrundprozesse (3 Punkte)

Implementieren Sie analog zum Kommando „`cd`“ aus Aufgabe 3.5 ein weiteres, internes Kommando namens „`jobs`“ in `execute_command()`, welches beim Aufruf die Prozess-ID und Kommandoeingabe aller aktuell laufenden Hintergrundprozesse auf der Standardausgabe ausgibt.

Für die Lösung dieser Aufgabe ist eine geeignete verkettete Liste mit dem Modul *plist* gegeben. Immer wenn ein Prozess im Hintergrund startet, sollen Sie diesen mittels `insert_element()` in die Liste aufnehmen. Analog dazu sollen Sie den Prozess mit `remove_element()` aus der Liste entfernen, wenn er terminiert.

Das Durchlaufen der verketteten Liste sollen Sie selbst in `plist.c` in folgender Funktion implementieren:

```
void walk_list( int(*callback)(pid_t, const char *) )
```

Die Funktion `walk_list()` erwartet als Parameter einen Funktionszeiger auf eine beliebige Funktion, die zwei Parameter vom Typ `pid_t` und `const char*` hat und einen Integer-Wert zurückgibt. Damit können Sie eine eigene, sogenannte callback-Funktion an `walk_list()` übergeben, welche die Textausgabe übernimmt. Implementieren Sie eine eigene callback-Funktion zum Ausgeben der Prozess-ID und der Aufrufzeile im *shellfunctions*-Modul.

3.7 Fehlerbehandlungen

Diese Aufgabe ist kontinuierlich neben den anderen Aufgaben zu lösen! Die Bewertung dieser Aufgabe ist in den anderen Teilaufgaben enthalten.

Achten Sie bei der Verwendung von Bibliotheksfunktionen wie **fork(3)**, **exec(3)**, **chdir(3)** oder **waitpid(2)** darauf, dass die Fehlerfälle geeignet behandelt werden. Dies kann beispielsweise das Fehlschlagen der Programmausführung oder des Verzeichniswechsels sein. Lesen Sie in den Man-Pages das Verhalten der Funktion im Fehlerfall nach und geben Sie die entstehende Fehlermeldung mittels **perror(3)** aus. Beachten Sie bitte, dass je nachdem welche Funktion fehlschlägt ein weiterer Betrieb der *myshell* nicht sinnvoll ist und der Prozess zusätzlich mit **exit(3)** beendet werden sollte. Wählen Sie dabei einen geeigneten *Exit Code*. **Nichtbeachten dieser Aufgabe kann zu einem Punktabzug von bis zu 4 Punkten führen.**

Abgabe bis 23.11.2018, 23:59 Uhr.

Ihre Lösungen sollen bis zur Deadline im GitLab des IBR hochgeladen sein.
Einhaltung der Coding Guidelines: 3 Punkte
Zu erzielende Minimalpunktzahl: 11 Punkte