Übungsblatt 4

Abgabe bis spätestens 28.11.2016 in Stud.IP

Aufgabe 1 (5 Punkte)

Schreibt eine einfache Shell, die in der Lage ist, Kommandos mit mehreren Argumenten einzulesen und entweder im Vordergrund oder im Hintergrund zu starten. Die Einleseroutine selbst (einschließlich Parsieren der Argumente) und ein Rumpf für das Hauptprogramm sind in `/home/ti2/ueb/04` vorgegeben. (Ihr benötigt für diese Übungsaufgabe das Programm `lex` bzw. `flex`, das möglicherweise nicht auf Eurem eigenen System vorinstalliert ist.)

Unser `ti2sh` verhält sich schon fast wie eine Shell, führt aber noch keine Kommandos aus. Erweitert die Vorgabe so, dass tatsächlich Prozesse erzeugt und in ihnen die eingelesenen Kommandos ausgeführt werden, und erläutert kurz Eure Vorgehensweise. (Ihr sollt nur die Datei `ti2sh.cc` ändern.) Hinweise dazu:

- Die typische Eingabesyntax eines Unix-Shellkommandos mit z. B. zwei Argumenten, das im Hintergrund gestartet werden soll, hat bekanntlich die Form:
  `kommando param1 param2 &`
- Die vorgegebene Einleseroutine parsiert die Kommandoeingabe und liefert eine Struktur `command` mit den Komponenten `argv` und `background` zurück.
  1. argv ist ein Array von Zeichenfolgen. Der erste Eintrag in diesem Array enthält den Kommandonamen, der zweite das erste Argument des Kommandos usw. (d. h. die „Worte“ der Eingabe werden von der Einleseroutine einzeln in das Array geschrieben). `cmd.argv[0]` indiziert den ersten Eintrag des Arrays.
  2. Wenn das angegebene Kommando nicht im Hintergrund ausgeführt werden soll, ist `background` null, andernfalls ist `background` 1.
- Die zu implementierende Shell soll keine Ein-/Ausgabeumlenkung (oder sonstige weitergehende Shell-Funktionalitäten) vorsehen. Auch müsst Ihr Euch explizit nicht um im Hintergrund laufende Prozesse kümmern, die von `STDIN` lesen wollen.
- Der Systemaufruf `fork()` erzeugt aus dem aktuellen Prozess heraus einen Kindprozess. Beide Prozesse durchlaufen zunächst dasselbe weitere Programm. Ein `fork()` liefert dem Vaterprozess die Prozess-ID des Kindes zurück und dem Kindprozess den Wert `0`.
- Systemaufrufe liefern im Fehlerfall oftmals `-1` zurück. Die Funktion `perror()` ermöglicht es dann, sinnvolle Fehlermeldungen auszugeben.
- Parsiert die über `getenv("PATH")` zu erlangende Environment-Variable `PATH`, um auf deren einzelne Elemente zugreifen zu können. Wenn Ihr ein Element von `PATH` mit dem Kommandonamen verknüpft habt, benutzt Ihr `execv()` zum Ausführen des eingelesenen Kommandos. Diese Funktion erwartet zwei Parameter:
  1. Den Pfadnamen, unter dem das auszuführende Kommando zu finden ist.

2. Das Array von Zeichenfolgen, das die einzelnen Worte der Eingabe enthält (`cmd.argv`). Es ist nur `execv()` zulässig, keine andere Variante von `exec()`.

- Es soll möglich sein, ein Kommando mit einem absoluten Pfad aufzurufen. Vergegenwärtigt Euch, was es bedeutet, einen Prozess in der Shell im Vordergrund bzw. Hintergrund zu starten, wie sich also Eure Shell zu verhalten hat und bedenkt, dass dies zunächst keine Auswirkungen auf den Zustand des Prozesses hat.
- Der Systemaufruf `wait()` zum Warten auf die Termination der von Eurer Shell erzeugten Kindprozesse liefert die Prozess-ID des terminierten Prozesses zurück und benötigt einen Parameter, den wir hier nicht nutzen wollen und deshalb auf `0` setzen (ein Nullpointer): `terminated_child = wait(0);`
  Alternativ könnt ihr `waitpid()` nutzen. Je nach Implementierungsansatz werdet Ihr einen Signal-Handler für das Signal `SIGCHLD` anmelden müssen, um tatsächlich alle Kinder einsammeln zu können.

Testet Eure Implementierung angemessen, d. h. nehmt Eingaben vor, die die geforderten Eigenschaften demonstrieren und dokumentiert dies.

Aufgabe 2 (2 Punkte)

Ein gedachter Rechner verwendet den Buddy-Algorithmus zur Speicherverwaltung. Anfangs existiert ein Block mit 64,KiB (65536 Byte) ab Adresse 0. Gebt für die folgenden Speicheranforderungen und ihre Auswirkungen jeweils eine Grafik analog zur Vorlesung bzw. Tutorium an:

a) Nachdem aufeinanderfolgende Anforderungen für 10 KiB, 12 KiB, 3 KiB, 16 KiB, 1 KiB, 20 KiB und 2 KiB eingetroffen sind: wieviele freie Blöcke bleiben übrig, was sind ihre Größen und Adressen?

b) Anschließend werden die Blöcke für die Anforderungen 2 KiB, 16 KiB und 3 KiB wieder freigegeben. Welche Blöcke gibt es nun? Könnte eine folgende Anforderung von 21 KiB erfüllt werden?

Aufgabe 3 (3 Punkte)

Gegeben sei die nachfolgende Abfolge von Shell-Kommandos zu den genannten abstrakten Zeitpunkten:

0	$ **bash**	
1	$ **ls**	
	fasel blub baz.c	
2	$ **xterm &**	
	[1] 12345	
3	$ **cc baz.c &**	
4		$ **make**
5		$ **ls -lR / &**
6	$ **kill -STOP 12345**	
	[1] Stopped xterm	
7	fg %1	
8	Ctrl-z	
	[1] Stopped xterm	
9	$ **bg**	
	[1]+ xterm &	
10		sleep 5
11	$ Ctlr-d	$

Ein `xterm` öffnet eine neue Shell in einem separaten Fenster des X-Window-Systems, genauso wie vergleichbare Programme auf Eurem jeweiligen Desktop. Im Beispiel laufen also zwei Shells nebenläufig zueinander.

Stellt die Änderungen der Prozesshierarchie über die angegebene Zeit grafisch dar wie im Tutorium gezeigt. Wann entstehen neue Prozesse, wann terminieren sie wieder? Wann wird welchem Prozess welches Signal gesendet (und von wem)? Zu welchem Prozess gehört am Ende das in Schritt 2 gestartete Terminal?

Geht vereinfachend davon aus, dass

- genügend Prozessoren für alle Prozesse zur Verfügung stehen, so dass keine Scheduling-Entscheidungen zu treffen sind und
- neue Kommandos sobald wie möglich und in Nullzeit eingegeben werden (d. h. der Zeitpunkt der Ausgabe eines Prompts und des Absendens des nächsten Kommandos ist derselbe).

Hinweis: „Ctrl“ ist auf deutschen Tastaturen mit „Strg“ beschriftet. „Ctrl-c“ und „Ctrl-z“ bezeichnet die Eingabe entsprechender Tastenkombinationen. Die Kommandos ‘`bg` und `fg` sind eingebaute Shell-Befehle, alle übrigen Kommandos sind eigenständige Programme.

Weitere Aufgaben

1. Warum werden die Zustandsinformationen eines Unix-Prozesses teilweise in der *Proc*-Struktur und teilweise in der *User*-Struktur abgelegt? Nenne jeweils drei charakteristische Beispiele für Angaben darin.
2. Skizziere kurz die Prozesserzeugung in Unix. Welche Rolle spielen die Systemaufrufe `fork()` und `exec()`?
3. Wie erfährt ein Unix-Prozess, ob ein Kindprozess terminiert ist? Wozu gibt es in Unix den Prozesszustand `SZOMB` („Zombie“)?
4. Welche Vor- und Nachteile hat der First-Fit- bzw. der Best-Fit-Algorithmus zur Speicherverwaltung? Wie arbeitet der Buddy-Algorithmus in etwa?

5. Wozu bieten Systeme eine Speicherhierarchie an? Welche Beobachtung über den Speicherzugriff realer Programme liegt dem zugrunde? Welche verschiedenen Arten von Speiche werden typischerweise bereitgestellt?
6. Warum ist es in der Regel nicht sinnvoll, den Adressraum eines Prozesses in einem Stück im Hauptspeicher abzulegen?
7. Was versteht man unter *Paging*, was unter *Segmentierung*? Wo tritt *interne Fragmentierung*, wo *externe Fragmentierung* auf? Was ist das?
8. Aus welchen Teilen besteht eine *virtuelle* Adresse zumeist? Wie ermittelt sich daraus die entsprechende Hauptspeicheradresse, d. h. wie läuft die Adressverwaltung in etwa ab?
9. Wie können mehrere Prozesse mit Hilfe virtueller Adressierung auf dieselben Programmstücke (oder auch Datenbereiche) zugreifen?

Diese Aufgaben müssen nicht abgegeben werden. Sie dienen als Vorbereitung auf das Fachgespräch und werden im Tutorium besprochen.

Abgabe

Bis 24:00 Uhr am 28.11.2016 digital in Stud.IP und – wenn nicht anders mit Eurem Tutor vereinbart – ausgedruckt in unser Postfach in der MZH-Ebene 6 oder im nächsten Tutorium. Es gelten die vereinbarten Scheinbedingungen (siehe Stud.IP). Bitte beachtet unsere ergänzenden Hinweise ebenda.

Packt die abgaberelevanten Dateien in eine Archivdatei. Die Dateinamen innerhalb des Archivs wie auch der Archivname selbst müssen den in den Scheinbedingungen festgelegten Namenskonventionen folgen. Abgaben, die diese Konventionen nicht einhalten, gelten als nicht erfolgt.

Eure Ansätze und der gewählte Lösungsweg müssen nachvollziehbar sein. Achtet insofern auf eine saubere Dokumentation im Quelltext. Benennt alle von euch verwendeten Quellen, auch Zusammenarbeit mit anderen Gruppen und verwendete Unterlagen aus früheren Jahrgängen.

Für Programmieraufgaben ist die Korrektheit der Lösung bzw. deren Grenzen grundsätzlich nachzuweisen. Dies geschieht neben der Dokumentation des Programmcodes durch geeignete Tests, deren Auswahl und Eignung begründet werden muss.