Lehrstuhl Connected Mobility
Fakultät für Informatik
Technische Universität München

Grundlagen von Betriebssystemen und Systemsoftware

WS 2018/19

Übung 7: Interprozesskommunikation

zum 3. Dezember 2018

- Die **Hausaufgaben** für dieses Übungsblatt müssen **spätestens** am **Sonntag, den 09.12.2018 um 23:59** in Moodle hochgeladen worden sein.
- Alle C-Programme müssen mit den folgenden Flags kompiliert werden:
  `gcc -Wall -o <progname> <prog.c>`.
- Hierzu stellen wir für jedes Übungsblatt jeweils ein Makefile bereit, das **nicht** verändert werden darf, um sicherzustellen, dass Ihre Abgabe auch korrekt von uns getestet und bewertet werden kann. **Ihr Programmcode muss zwingend mit dem Makefile kompilierbar sein. Anderenfalls kann die Abgabe nicht bewertet werden.** Wenn Sie zwischenzeitlich Änderungen vornehmen wollen, um etwa bestimmte Teile mittels `#ifdef` einzubinden und zu testen, dann kopieren Sie am besten das Makefile und modifizieren Ihre Kopie. Mit `make -f <Makefile>` können Sie dann eine andere Datei zum Übersetzen verwenden.
- Die Abgabe der Programme erfolgt als Archivdatei, die die verschiedenen Quelldateien (.{c|h}) umfasst und **nicht** in Binärform. Nicht kompilierfähiger Quellcode wird **nicht gewertet**.
- Um die Abgabe zu standardisieren enthält das Makefile ein Target "submit" (`make submit`), was dann eine Datei blatt07.tgz zum Hochladen erzeugt.
- Damit die richtigen Dateien hochgeladen und ausgeführt werden, geben wir bei allen Übungen die jeweils zu verwendenden Dateinamen für den Quellcode und auch für das Executable an.
- Die Tests werden auf diesem Aufgabenblatt mittels Python-Programmen durchgeführt. Zum Ausführen der Tests geben Sie `python xyz_test.pyc` auf der Konsole ein. Ihre ausführbaren Programme müssen sich im selben Verzeichnis wie die Testprogramme befinden.

1 Vorbereitung auf das Tutorium

Was ist die Rolle der folgenden POSIX- (Portable Operating System Interface) bzw. Bibliotheksfunktionen? Erläutern Sie kurz die Parameter und den Rückgabewert jeder Funktion. [1]

- `int open (const char *path, int oflag, ...);`
- `int pipe (int *filedes);`
- `int dup (int filedes);`
- `int dup2 (int filedes, filedes2);`
-

2 Tutoraufgaben

2.1 Parallele Modellierung – Vom Modell zur Anwendung

Eine Fabrik stellt verschiedene Arten von Möbeln her. Diese Möbel werden in einem Lager mit begrenzter Kapazität gelagert. Die gelagerten Möbel werden von Mitarbeitern zur Laderampe geschleppt, von wo sie an Händler ausgefahren werden. Die Rampe hat ebenfalls eine begrenzte Kapazität. Das Ausfahren der Ware zum jeweiligen Geschäft übernimmt je ein Lastwagen. In den Geschäften werden die Möbel letztendlich an Kunden verkauft.

- Modellieren Sie für das gegebene Szenario ein Petrinetz mit folgenden Randbedingungen:
  - Die Fabrik produziert **Betten** und **Schränke**.
  - Die Kapazität des Lagers beträgt **500 Möbelstücke**.
  - **Ein Mitarbeiter** übernimmt den Transport von Möbelstücken aus dem Lager zur Rampe.
  - Auf der Rampe können **10 Möbelstücke** lagern.
  - **Zwei Lastwagen** fahren Möbelstücke aus. Es passt nur ein Möbelstück pro Fahrzeug.
  - **Zwei Geschäfte** nehmen die Ware ab
    - Das erste Geschäft verkauft nur Betten. Es können maximal **30 Betten** dort gelagert werden.
    - Das zweite Geschäft verkauft sowohl Betten als auch Schränke. Es kann je **20 Betten** und **20 Schränke** aufnehmen.

  Anregungen zur Konstruktion des Petrinetzes:

  - Überlegen Sie sich eine geeignete Realisierungsmöglichkeit für das zugrunde liegende Producer-Consumer Problem (Fabrik produziert, Kunden konsumieren) in einem Petrinetz. Denken Sie dabei an die Gewichtung von Kanten (insbesondere auch 0).
  - Steuern sie die Transportvorgänge anhand der Kapazitäten an den verschiedenen Orten. Wo muss die gesamte Anzahl der Möbelstücke, wo die Anzahl einzelner Betten bzw. Schränke gesteuert werden? Wo nicht?

[1] Erläuterungen und Beispiele der einzelnen Funktionen finden Sie auch in den **man**-Pages auf Ihrer Linux-VM.

  - Verwenden Sie die in der Vorlesung eingeführten Kontrollstrukturen, um sicherzustellen, dass z.B. nicht mehr Betten bzw. Schränke als tatsächlich vorhanden ausgeliefert werden.
  - Achten Sie auf die Transportmodellierung. Erlauben Sie nicht das Aufsammeln von neuen Waren, wenn die alten noch nicht ausgeliefert wurden (ein Transporter kann nicht gleichzeitig am Start und am Ziel sein).
  - Überprüfen Sie Ihre Modellierung. Das zu konstruierende Petrinetz ist komplex, wenn es alle Anforderungen abdeckt[2]. Machen Sie sich mit den mannigfaltigen Modellierungsmöglichkeiten vertraut, die Ihnen ein Petrinetz bietet.

- *Diskussion:* Identifizieren Sie *nebenläufige* Bereiche und überlegen Sie sich, ob das modellierte Petri-Netz Transitionen besitzt, die verhungern könnten.

2.2 Petri-Netze und Erreichbarkeit

Gegeben sei das Petri-Netz in Abbildung 1. Wir wollen *zunächst* einmal annehmen, dass es sich dabei um ein boolsches Petri-Netz (d.h. es kann maximal nur eine Marke auf einer Stelle stehen) handelt.

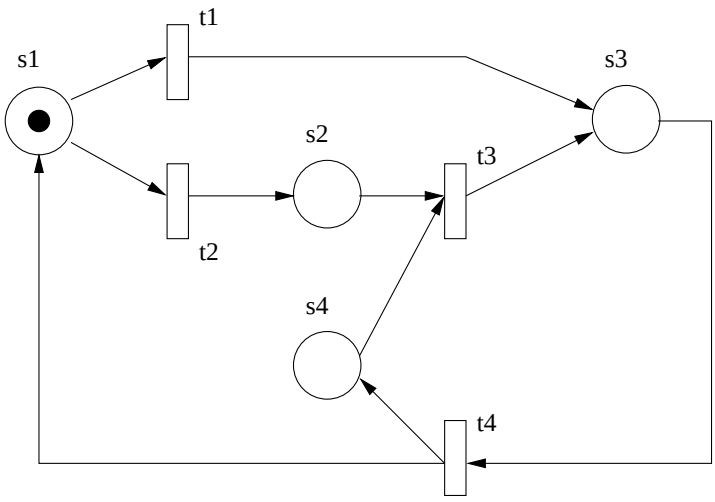

Figure 1: Petri-Netz

2.2.1 Erreichbarkeit

Welche Belegungen sind erreichbar? Geben Sie den Erreichbarkeitsgraphen an.

2.2.2 Nebenläufigkeit

Existiert ein nicht-sequentieller Ablauf? Falls ja, geben Sie ein Beispiel an. Falls nein, begründen Sie Ihre Antwort.

2.2.3 Verklemmungen

Ist eine Verklemmung erreichbar? Begründen Sie Ihre Antwort.

[2]Zum Vergleich: die Mustermodellierung hat 15 Stellen

2.2.4 Modifizierung des Modells

Ändern sich die möglichen Abläufe, wenn nicht nur boolsche, sondern auch natürlichzahlige Belegungen zugelassen werden?

Bitte beachten Sie: Die Ausgangsbelegung des Netzes bleibe dabei gleich.

2.2.5 Starvation/Verhungern

Ist bei den natürlichzahligen Belegungen aus Teilaufgabe 2.2.4 ein Verhungern einer Transition möglich? Falls ja, geben Sie einen Ablauf an, bei dem eine Transition ausgehungert wird. Falls nein, begründen Sie Ihre Antwort.

2.3 Und noch ein Petrinetz

Gegeben sei folgendes boolesches Petri-Netz (S, T, F):

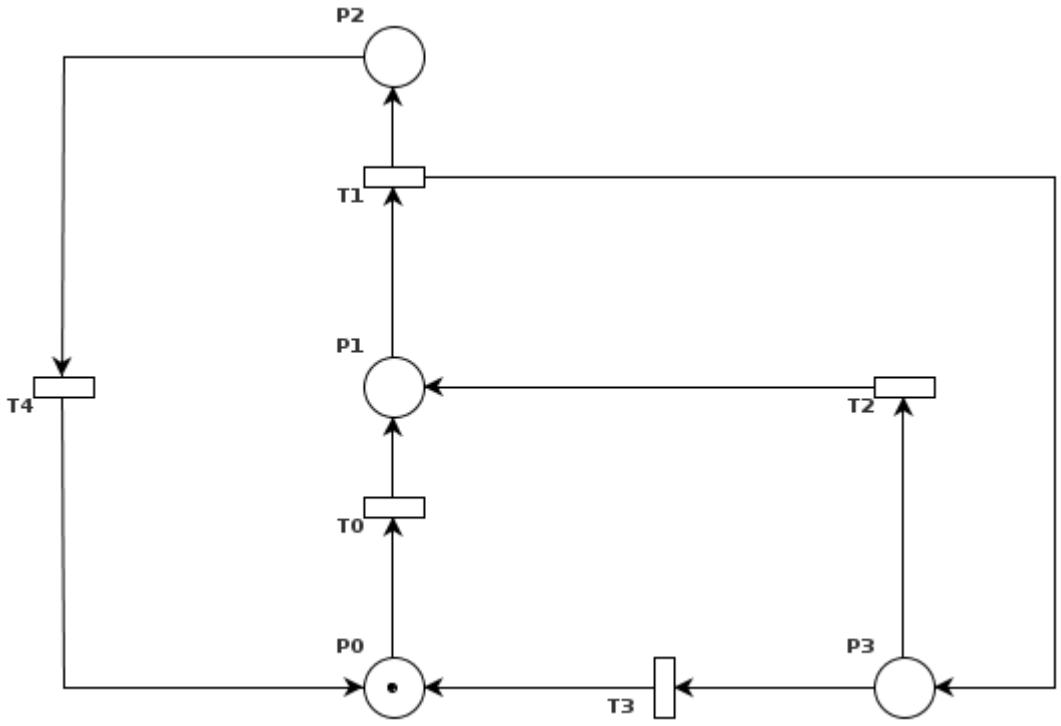

Figure 2: Petri-Netz

a) Geben Sie den Erreichbarkeitsgraphen an.

b) Argumentieren Sie mit Hilfe des Erreichbarkeitsgraphen, ob das Netz verklemmt ist.

c) Beseitigen Sie eine ggf. vorhandene Verklemmung durch Einführen einer neuen Stelle (mit Kapazität 1) und entsprechenden Transitionen.

2.4 Ein-/Ausgabe: Kopieren einer Datei

Zur Vorbereitung auf die Hausaufgaben wollen wir ein einfaches Kommando `mycp` schreiben. Dieses Kommando kopiert einen Eingabestrom auf einen Ausgabestrom. Wenn kein Kommandozeilenparameter angegeben wird, kopiert das Programm alles, was es von der Standardeingabe liest, auf die Standardausgabe.

Zwei Optionen sollen für die Kommandozeile vorgesehen werden:

- `-i Dateiname` liest von der angegebenen Datei statt von der Standardeingabe.
- `-o Dateiname` schreibt in die angegebene Datei statt auf die Standardausgabe.

Je nach verwendetem Ein- sowie Ausgabestrom wird nach dem Öffnen aller Dateien mittels "`Copying x -> y`" angegeben, von welchem Dateideskriptor gelesen (x) und auf welchen Dateideskriptor (y) geschrieben wird (siehe Beispiel).

Die Standardeingabe hat den Dateideskriptor 0, die Standardausgabe den Deskriptor 1 (und die Ausgabe für Fehlermeldungen, wie sie etwa von `perror()` ausgegeben werden hat den Deskriptor 2).

Beispiele:

```
$ ./mycp
Copying 0 -> 1
Hello
Hello
World
World
$ ./mycp -o xyz.txt
Copying 0 -> 3
Hier erzeugen wir mal etwas Inhalt fuer einen Testdatei.
Damit wir die dann kopieren koennen.
$ ./mycp -i xyz.txt -o abc.txt
Copying 3 -> 4
$ ./mycp -i abc.txt
Copying 3 -> 1
Hier erzeugen wir mal etwas Inhalt fuer einen Testdatei.
Damit wir die dann kopieren koennen.
```

3 Hausaufgaben

Auf diesem Aufgabenblatt wollen wir Ihre Shell um zwei Funktionen ergänzen, die sich mit Interprozesskommunikation und grob verwandten Konzepten beschäftigen.

3.1 Umleitung der Ein-/Ausgabe (7 Punkte)

Wir nähern uns der Interprozesskommunikation mittels Pipes in einem ersten Schritt an, indem wir uns mit der Umleitung der Standardein- und -ausgabe beschäftigen. In einer Shell werden hierzu die Symbole `'<'` und `'>'` verwendet. Diese sollen Sie nun in der Kommandozeile Ihrer Shell erkennen. Achten Sie darauf, dass jeder der beiden Parameter nur maximal einmal vorkommen darf, denn sonst wäre nicht klar, in welche Datei geschrieben bzw. welche Datei gelesen werden sollte.

Wichtig: Sie **können** hierzu Ihren Parser vom Blatt 06 erweitern. Sie **können aber auch** einen vereinfachten neuen Parser schreiben, der keine Sonderzeichen (`'"$\`) unterstützen muss. Dieser muss lediglich einen Eingabestring in durch ein oder mehrere Leerzeichen getrennte Parameter transformieren. Egal ob alt oder neu, verwenden Sie hierfür wieder die Datei parser.c.

- `< Dateiname` liest statt von der Standardeingabe aus der Datei mit dem Namen "Dateiname".
- `> Dateiname` schreibt statt auf die Standardausgabe in die Datei mit dem Namen "Dateiname".

Das Umleiten der Standardein- und/oder -ausgabe muss der von der Shell mit `fork()` erzeugte Kindprozess umsetzen, denn Sie wollen ja nicht die Ein-/Ausgabe der Shell ändern, sondern nur die des jeweiligen Kommandos. Nach dem Umsetzen können Sie dann das Kommando mit `execve()` ausführen.

Die Standardeingabe hat den Dateideskriptor 0, die Standardausgabe den Deskriptor 1 (und die Ausgabe für Fehlermeldungen, wie sie etwa von `perror()` ausgegeben werden hat den Deskriptor 2). Alle Dateideskriptoren werden in einer Tabelle verwaltet, und 0, 1, 2 usw. sind Indizes in dieser Tabelle. Wird eine neue Datei geöffnet oder eine Pipe erzeugt, so wird der niedrigste freie Index verwendet. Wenn Sie also erst die Standardeingabe schließen (`close(0)`) und danach eine Datei öffnen wird für diese neu geöffnete Datei der Deskriptor 0 (wieder-)verwendet. Auf diese Weise können Sie gezielt die Ein-/Ausgabe umlenken. Sie können eine Datei auch erst öffnen und anschließend mit dem Systemaufruf `dup()` diesen Dateideskriptor duplizieren, nachdem Sie Standardein- und/oder Standardausgabe geschlossen haben – auch dann wird der niedrigste freie Eintrag in der Deskriptortabelle verwendet.

Hinweise: Sie können Ihre Kommandozeileninterpretation in zwei Stufen realisieren. Die erste trennt den Eingabestring in mehrere Parameter auf, inklusive ">" und "<" (und dann "|" in der nächsten Aufgabe) und liefert alle diese Parameter in einer Liste zurück, wie bei der letzten Aufgabe auch. Die zweite Stufe arbeitet dann nur noch auf der Liste und sucht nach den Sonderzeichen für die Umleitung der Ein-/Ausgabe, entfernt diese und die zugehörigen Parameter aus der Liste (und stellt auch fest, wenn ein Parameter fehlt bzw. ein das gleiche Sonderzeichen zwei Mal auftritt) führt dann die entsprechende Sonderbehandlung durch.

Sie brauchen, wie oben schon erwähnt, das Symbol "\" hier nicht zu berücksichtigen und müssen sich insbesondere **keine** Gedanken machen, wie Sie in bei einer zweitstufigen Interpretation der Eingabestrings etwa die Zeichenfolgen "\|". "\<" pder "\>" korrekt behandeln.

Beispiel:

```
$ ls > files.txt
$ cat files.txt
Makefile
files.txt
list.c
list.h
mysh
mysh.c
parser.c
$ wc < files.txt
       7       7      54
```

Quellen: parser.c mysh.c list.c list.h

Executable: mysh

3.2 Pipes für die Kommandozeile Ihrer Shell (3 Punkte)

Sie haben in der Vorlesung den System-Call `pipe()` kennengelernt. In dieser Aufgabe sollen Sie die Funktion solcher Pipes in Ihre Shell einbauen. Die grundsätzliche Vorgehensweise ist dabei ganz ähnlich wie bei der Umleitung von Standardein- und -ausgabe, nur dass Sie jetzt nicht aus einer bzw. in eine statische Datei umleiten, sondern in eine Pipe unter Verwendung des entsprechenden Kommandos `pipe()`. Erinnern Sie sich daran, dass die Kommunikation in den Pipes in einer Shell strikt unidirektional ist.

Erweitern Sie hierzu Ihre Shell aus der vorigen Aufgabe. Für die Realisierung einer Pipe verwenden wir das Pipe-Symbol ("|"). Sie müssen nicht mehr als ein Pipe-Symbol auf der Kommandozeile unterstützen. Ihre Shell soll auch weiterhin die Umleitung von Ein- und Ausgabe unterstützen.

```
$ cat mysh.c parser.c | grep -n else
```

In obigem Beispiel werden zwei Kommandos erzeugt:

- `cat mysh.c parser.c` gibt die Inhalte der beiden Dateien `mysh.c` und `parser.c` nahtlos nacheinander auf der Standardausgabe aus.
- `grep else` durchsucht den von der Standardeingabe gelesenen Daten nach Zeilen, die das Suchwort "else" enthalten und gibt nur noch diese Zeile aus. Die Option "-n" sorgt dafür, dass die laufende Zeilennummer vorangestellt wird.

Die Shell muss also für jedes Auftreten einer Pipe in der Kommandozeile jeweils einen weiteren Prozess starten. Vor dem Erzeugen der Kindprozesse muss der Elternprozess jeweils dafür sorgen, dass deren Standardein- und -ausgabe entsprechend in Serie durch vom Elternprozess erzeugte Pipes miteinander verbunden sind.

Hinweise: In dieser Aufgabe müssen Sie beim Auftreten einer Pipe zwei Kindprozesse erzeugen, einer für den Teil links und einen für den Teil rechts von der Pipe. (Im allgemeinen Fall wären es $n+1$ Prozesse bei n Pipes.) Für jeden müssen Sie seinen Teil der Liste in ein Array umwandeln, wie Sie es auch auf dem letzten Aufgabenblatt bereits getan haben (dort allerdings nur für einen Prozess).

Das Erzeugen der Pipe muss im Shell-Prozess geschehen, bevor das erste Kommando erzeugt wird, denn die Deskriptoren dieser Pipe können nur mittels `fork()` an die beiden Kind-Prozesse weitergegeben werden. Der Shell-Prozess darf die Pipe erst schließen, wenn beide Kindprozesse erzeugt worden sind. Dann muss(!) er das aber auch tun!

Denken Sie daran, dass ein Prozess das Ende einer Eingabe erst dann erkennt, wenn alle möglichen Quellen für die Standardeingabe geschlossen worden sind. Sie müssen also dafür sorgen, dass Sie alle nicht benötigten Verweise auf den Deskriptor schlieẽen.

Achten Sie darauf, dass ein Prozess nur entweder aus einer Pipe lesen kann oder aus einer Datei mittels Eingabeumleitung. Ebenso kann ein Prozess seine Standardausgabe nur entweder über eine Pipe dem Folgeprozess zur Verfügung stellen oder die Ausgabe in eine Datei umleiten lassen.

Wenn ein Kindprozess sein Kommando nicht ausführen kann, sollte er die anderen Kindprozesse per Signal beenden (SIGKILL). Starten Sie Ihre Kindprozesse entsprechend der Kommandozeile von links nach rechts. Vermerken Sie die Prozess-IDs in einer Datenstruktur des Shell-Prozesses, die dann ja jeweils an alle Kindprozesse vererbt wird. Wenn Sie den n-ten Prozess starten und etwas schiefgeht, können Sie einfach die Datenstruktur bis $n-1$ durchgehen und die anderen Prozesse beenden.

Quellen: parser.c mysh.c list.c list.h

Executable: mysh

Ein Beispiel (natürlich hängt die Ausgabe in diesem Fall von Ihrem Code ab). Lassen Sie sich nicht durch die Zeilennummern irritieren, denn die Musterlösung ist allgemeiner gehalten und kann beispielsweise mehr als zwei Pipes pro Kommandozeile ausführen.

```
$ ./mysh
$ cat mysh.c parser.c | grep -n else
53:    } else
106:                else
114:        } else if (!strcmp (le->data, "<") || !strcmp (le->data, ">")) {
122:                    else {
128:                } else {
132:                    else {
140:            } else {
147:        } else {
180:            } else {
230:            } else {
480:        } else {
556:            else {
$ exit
```