Praktikumsaufgabe 1 - Programme & Systemaufrufe

In diesem Praktikum implementieren Sie eine Erweiterung für das Shell-Programm des Betriebssystems, sowie das Kommandozeilenprogramm find und die Systemaufrufe lseek und sysinfo. Des Weiteren werden Sie sich mit Systemaufrufverfolgung beschäftigen.

Programme

Aufgabe 1 Erweiterung Shell-Programm (4 Punkte)

Betrachten Sie die Implementierung der Shell in der Datei `user/sh.c`. Ihre Aufgabe ist es, das bestehende Programm so zu erweitern, dass die Shell unabhängig vom aktuellen Arbeitsverzeichnis, auch immer im root-Verzeichnis `'/'` nach dem ausführbaren Programm sucht. Sollte es zwei Programme mit dem gleichen Namen im aktuellen Verzeichnis als auch im root-Verzeichnis geben, dann soll das Programm im aktuellen Verzeichnis ausgeführt werden. Unbahängig davon, ob sich ein Programm im root-Verzeichnis befindet, soll das Arbeitsverzeichnis unverändert bleiben. Sollte ein Programm weder im aktuellen noch im root-Verzeichnis sein, geben Sie hierfür eine spezfische Fehlermeldung aus.

Hinweise

- Machen Sie sich soweit mit dem Shell-Programm vertraut, dass Sie folgende Fragen beantworten können:
  - Zwischen welchen Input Typen wird unterschieden?
  - Welche Schritte werden ausgeführt, um ein anderes Programm zu starten?
  - Wie werden Programme, die ausgeführt werden sollen, gefunden?
- Nutzen Sie Filedeskriptoren zum Prüfen, ob eine Datei existiert.
- Nutzen Sie die String Funktionen aus `user/ulib.c`.
- Für die Fehlerausgabe: Was ist der Unterschied zwischen `printf`und `fprintf` und welches sollte für Fehlermeldungen verwendet werden?
  - Betrachten Sie hierfür die Standard I/O und Error Deskriptoren eines Prozesses.

Aufgabe 2 find (4 Punkte)

Für diese Aufgabe sollen Sie ein eigenes Kommandozeilenprogramm in C Schreiben. Das Programm soll `find` heißen und ausgehend vom aktuellen Verzeichnis, dieses und alle Unterverzeichnisse nach einem gegeben Namen absuchen. Alle Dateien oder Verzeichnisse, die den gesuchten Namen in ihrem

Namen enthalten, sollen zusammen mit der Typ Nummer ausgegeben werden. Die Suche soll case-sensitive sein. `find` soll 1 bis n Argumente akzeptieren und einzeln nach jedem übergebenen Namen suchen. Arbeiten Sie mit Fehlerausgaben über `fprintf` (z.B. wenn kein Argument an `find` übergeben wurde).

Beispiel: Es wird nach test und ls gesucht. Die führende Nummer gibt den Typ an (Datei oder Ordner...)

```
$ find test ls
2    ./forktest
2    ./usertests
2    ./lseektest
2    ./sysinfotest
1    ./test
2    ./test/BLAtest
2    ./ls
2    ./lseektest
```

Hinweise

- Erstellen Sie die Datei `user/find.c` und fügen Sie diese in das Makefile ein.
  - Betrachten Sie hierfür, wie die anderen Anwendungen eingebunden wurden.
- Betrachten Sie die Implementierung der anderen Programme, vor allem von `ls` als Ausgangspunkt.
- Machen Sie sich mit der grundlegenden Struktur der Implementierung des Dateisystems in der Datei `kernel/fs.h` vertraut.
- Nutzen Sie die String Funktionen aus `user/ulib.c`. Dort fehlt eine Funktion, um eine Substring in einem String zu finden. Implementieren Sie diese selber.
- Achten Sie auf die Ordner `.` und `..`

Systemaufrufe

Aufgabe 1 lseek (4 Punkte)

Machen Sie sich mit dem Systemaufruf `lseek(int fd, int offset, int whence)` vertraut, indem Sie dessen man page lesen. Finden Sie heraus, was der Systemaufruf macht und welche Parameter dieser benötigt. Implementieren Sie anschließend den Systemaufruf im xv6 Betriebssystem. Es reicht aus, wenn für den Parameter `whence` `SEEK_SET`, `SEEK_CUR` und `SEEK_END` akzeptiert werden. Mit anderen Werten soll der Systemaufruf fehlschlagen.

Implementieren Sie weiterhin ein Benutzerprogramm `lseektest.c`, mit dem Sie den Systemaufruf testen können. Überlegen Sie sich sinnvolle Testfälle.

Hinweise

- Machen Sie den Systemaufruf im User mode bekannt, indem Sie entsprechende Einträge in `user/user.h` und `user/usys.pl` hinzufügen.
- Definieren Sie den Systemaufruf im Kernel, indem Sie entsprechende Einträge in `kernel/syscall.h` und `kernel/syscall.c` hinzufügen.
- Definieren Sie `SEEK_SET`, `SEEK_CUR` und `SEEK_END` in `kernel/fcntl.h`.
- Implementieren Sie den Systemaufruf `uint64 sys_lseek(void)` in `kernel/sysfile.c`.
- Verhindern Sie, dass der Offset auf Werte kleiner null und größer als die Dateigröße gesetzt werden kann.
- Schauen Sie sich als Hilfestellung die Implementierung anderer Systemaufrufe in `kernel/sysfile.c` und das `struct file` in `kernel/file.h` an.

Aufgabe 2 sysinfo (4 Punkte)

In dieser Aufgabe implementieren Sie den Systemaufruf `sysinfo(struct sysinfo *)`. Mit diesem Aufruf kann ein Programm Information über das System abrufen. Im Falle des xv6 Betriebssystems sollen dies der freie Arbeitsspeicher und die Anzahl der laufenden Prozesse sein.

Für die Aufgabe wurden die zwei Dateien `kernel/sysinfo.h` und `user/sysinfotest.c` in das Repository hinzugefügt, die Sie im jeweiligen Verzeichnis finden. Die Datei `sysinfo.h` enthält eine Definition für `struct sysinfo`, welche dem Systemaufruf mit übergeben wird und von diesem mit Informationen gefüllt werden soll. `user/sysinfotest.c` ist ein Benutzerprogramm, mit dem Sie ihre Implementierung testen können. Ihre Implementierung ist korrekt, wenn das Programm den folgenden Text ausgibt:

```
sysinfotest: start
Free mem test: Got 0, expected 0
Free mem test: Got 133132288, expected 133132288
Free mem test: Got 133128192, expected 133128192
Free mem test: Got 133132288, expected 133132288
N proc test: Got 4, expected 4
N proc test: Got 3, expected 3
sysinfotest: OK
```

Hinweise

- Machen Sie den Systemaufruf im User mode bekannt, indem Sie entsprechende Einträge in `user/user.h` und `user/usys.pl` hinzufügen.
- Definieren Sie den Systemaufruf im Kernel, indem Sie entsprechende Einträge in `kernel/syscall.h` und `kernel/syscall.c` hinzufügen.

- Implementieren Sie in `kernel/kalloc.c` eine Funktion, die die Menge an freiem Arbeitsspeicher in Bytes zurückgibt.
- Implementieren Sie in `kernel/proc.c` eine Funktion, die die Anzahl der laufenden Prozesse zurückgibt. (Prozesse gelten in diesem Fall als "Laufend", wenn sie nicht den Status `UNUSED` besitzen.)
- Definieren Sie Ihre Funktionen aus `kernel/kalloc.c` und `kernel/proc.c` in `kernel/defs.h`.
- Implementieren Sie den Systemaufruf `uint64 sys_sysinfo(void)` in der Datei `kernel/sysproc.c`.
- In `sys_sysinfo` muss der Parameter vom Kernel space in den User space kopiert werden. Nutzen Sie dafür die Funktion `copyout()`. Beispiele bezüglich der Verwendung finden Sie in den Funktionen `sys_fstat()` (`kernel/sysfile.c`) und `filestat()` (`kernel/file.c`).
- Erstellen Sie im Makefile einen Eintrag für das Benutzerprogramm `user/sysinfotest.c`.

Aufgabe 3 Systemaufrufverfolgung (4 Punkte)

Systemaufrufverfolgung oder auf Englisch System call tracing ist nützlich, wenn man herausfinden möchte, welche Systemaufrufe ein Programm nutzt. In dieser Aufgabe implementieren Sie den Systemaufruf `trace(int tracemask)`, welcher die Systemaufrufverfolgung startet. Bei nachfolgenden Systemaufrufen sollen Informationen in dem folgenden Format ausgegeben werden:

```
{PID}: syscall {NAME} -> {RETURN_VALUE}
```

Die Angaben in den geschweiften Klammern sind mit entsprechenden Informationen zu ersetzen. Eine Ausgabe der übergebenen Parameter ist nicht nötig. Binden Sie zum Testen das von uns zur Verfügung gestellte Programm `trace` ein. Die Ausgaben des Testprogramms sollten bei den folgenden Befehlen mit den hier gezeigten Ausgaben übereinstimmen. Es kann passieren, dass sich die PIDs bei mehrfachem Ausführen ändern:

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
$
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 968
4: syscall read -> 235
4: syscall read -> 0
4: syscall close -> 0
$
```

```
$ trace 2 usertests forkfork
usertests starting
5: syscall fork -> 6
test forkfork: 5: syscall fork -> 7
7: syscall fork -> 8
7: syscall fork -> 9
8: syscall fork -> 10
9: syscall fork -> 11
8: syscall fork -> 12
9: syscall fork -> 13
[...]
$
```

Hinweise

- Fügen Sie dem `struct proc` in `kernel/proc.h` das Attribut `int tracemask` hinzu.
- Implementieren Sie in der Datei `kernel/sysproc.c` den Systemaufruf `uint64 sys_trace(void)`, welcher das Attribut `tracemask` des aktuellen Prozesses verändert.
- `tracemask` ist eine Maske, die angibt, welche Systemaufrufe ausgegeben werden sollen. (Beispiel: Für den Systemaufruf `read` mit der ID 5 wird die Maske `0b100000` verwendet.)
- Aktualisieren Sie die Funktion `void syscall(void)` in der Datei `kernel/syscall.c`, sodass bei jedem Aufruf eine entsprechende Ausgabe erfolgt, wenn die Systemaufrufverfolgung aktiviert ist.
- Stellen Sie sicher, dass bei einem Aufruf von `fork()` der Kindprozess das Attribut `tracemask` vom übergeorneten Prozess übernimmt.
- Erstellen Sie im Makefile einen Eintrag für das Benutzerprogramm `user/trace.c`.