Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
WiSe 2010/2011

AG Betriebssysteme FB3
Jan Peleska, Florian Lapschies

Abgabe: 16.02.2011

Übungszettel 6

User-Space Scheduling mit nicht-blockierendem I/O

Beim User-Space Scheduling müssen Threads freiwillig die CPU abgeben, damit andere User-Space Threads rechnen können. Aus diesem Grund sind blockierende Funktionen mit Vorsicht zu verwenden, da durch die Blockierung des aktuellen User-Space Threads alle anderen Threads ebenfalls blockiert werden. Glücklicherweise gibt es die Möglichkeit nicht-blockierende IO-Operationen durchzuführen. Im letzten Übungszettel habt ihr bereits `select()` verwendet, um im Voraus zu erfragen, ob ein nachfolgendes `read()` blockieren würde. Alternativ kann ein File-Deskriptor jedoch auch als nicht-blockierend markiert werden, so dass Operationen auf diesem anstatt zu blockieren mit entsprechenden Fehlern wie `EAGAIN` oder `EWOULDBLOCK` zurück kehren.

In diesem Übungszettel soll eine Anwendung `merge` in zwei User-Space Threads auf dem selben LWP Daten von jeweils einem eigenen nicht-blockierenden TCP/IP Socket empfangen. Als Daten bekommen die beiden Sockets lexikalisch aufsteigend sortierte Wörter von zwei gleichartigen Anwendungen `send` gesendet. Die Aufgabe von `merge` ist es, in den beiden User-Space Threads jeweils Sequenzen von sortierten Wörter zu empfangen und diese dann gemeinsam gebündelt sortiert auszugeben.

Aufgabe 1: Die `merge`-Anwendung

`merge` bekommt die beiden Port-Nummern, von denen die Sockets Daten empfangen sollen als Aufrufparameter übergeben. Sorgt dafür dass `merge` gegenüber falschen Aufrufparametern robust ist. Bei falschem Aufruf sollen entsprechende Fehlermeldung ausgegeben werden. Zum Auslesen der Portnummern aus dem Argumentvektor benutzt ihr `strtol()`.

Erstellt zwei TCP/IP Sockets, die ihr an jeweils einen der übergebenen Ports bindet. Sammelt dann mit `listen()` und einem Backlog der Größe eins eingehende Verbindungen. Die Sockets sollen als nicht-blockierend eingestellt werden. Benutzt hierzu `fcntl()` mit `F_GETFL`, um die aktuellen Flags der Socket File-Deskriptoren zunächst auszulesen. Fügt diesen Flags dann `O_NONBLOCK` hinzu und setzt diese mit `fcntl()` und `F_SETFL`.

Startet zwei User-Space Threads, die beide abwechselnd geschedult werden. Die Threads sollen zunächst mit `accept()` auf eine Verbindung warten. Damit `accept()` genauso angewendet werden kann wie ein `accept()` auf ein blockierenden Socket, schreibt ihr euch einen Wrapper `ut_sched_accept()` mit der gleichen Signatur wie `accept()`. Dieser ruft `accept()` in einer Schleife so lange auf, wie `EAGAIN` oder `EWOULDBLOCK` als Fehler zurück gegeben werden. Bei jedem Schleifendurchlauf wird die CPU freiwillig abgegeben, damit der andere Thread nicht blockiert wird. `ut_sched_accept()` gibt den Rückgabewert des letzten Aufrufs von `accept()` zurück. Die von `ut_sched_accept()` zurück gegebenen File-Deskriptoren setzt ihr wie oben beschrieben ebenfalls auf nicht-blockierend.

Nachdem eine Verbindung aufgebaut wurde, empfangen die Threads mit `read()` Daten. Auch hier soll `read()` nicht direkt, sondern über einen Wrapper `ut_sched_read()` aufgerufen werden. `ut_sched_read(fd, buf, count)` liest in einer Schleife von `fd` nach `buf`, bis die

gesamten `count` Bytes gelesen wurden oder das Ende der Datei erreicht wurde. Nach jedem Schleifendurchlauf wird die CPU freiwillig abgegeben, damit der andere Thread nicht blockiert wird. Treten andere Fehler als `EAGAIN` und `EWOULDBLOCK` auf, wird die Schleife verlassen und `-1` zurück gegeben. Ansonsten wird am Ende entsprechend der Semantik von `read()` die Anzahl der gelesenen Bytes zurück gegeben.

Die Threads empfangen nun in einer Schleife mit `ut_sched_read()` einzelne Wörter. Bei dem hierbei zu verwendenden Protokoll wird zunächst ein Längenfeld vom Typ `uint32_t` übertragen. Direkt danach folgt das Wort mit der entsprechenden Länge in dem Stream. Um ein einzelnes Wort auszulesen, muss also in einem ersten Schritt die Länge gelesen werden und dann in einem zweiten Aufruf von `ut_sched_read()` das eigentliche Wort. Achtet beim Lesen und Schreiben des Längenfeldes auf korrektes Network-Byteordering.

Nach dem Lesen eines einzelnen Wortes, wird dieses mit dem anderen Thread gebündelt lexikalisch aufsteigend sortiert auf `stdout` ausgegeben. Hierzu befindet sich der Thread in einer Schleife, bis er das gelesene Wort ausgeben konnte. Das eigene Wort wird hier mit `strcmp()` mit dem Wort des anderen Threads verglichen. Ist das eigene Wort das lexikalisch kleinere oder das gleiche wird es ausgegeben, um dann das nächste Wort zu empfangen. Ist das eigene Wort lexikalisch größer wird die CPU an den anderen Thread abgegeben. Wenn der andere Thread bereits alle seine empfangenen Wörter geschrieben hat, kann das eigene Wort sofort geschrieben werden. Wenn der andere Thread noch gar nichts lesen konnte, weil dieser noch im Verbindungsaufbau ist, wird die CPU freiwillig abgegeben.

Nachdem ein Wort schließlich ausgegeben wurde, wird die die CPU freiwillig abgegeben und bei der nächsten Aktivierung das nächste Wort verarbeitet. Sollte beim Lesen vom Socket das Ende der Datei erreicht sein, beendet sich der Thread.

Wenn sich beide Threads beendet haben, gibt der User-Space Scheduler die Kontrolle zurück und das Programm beendet sich.

Aufgabe 2: Die `send`-Anwendung

Die `send` Anwendung hat die Aufgabe, lexikalisch aufsteigend sortierte Wörter an einen der beiden Ports von `merge` zu senden.

`send` bekommt den Netzwerknamen und den Port von `merge` durch Doppelpunkt getrennt als Aufrufparameter übergeben. Sorgt dafür dass `send` gegenüber falschen Aufrufparametern robust ist. Bei falschem Aufruf sollen entsprechende Fehlermeldung ausgegeben werden.

Benutzt `getaddrinfo()`, um die zu dem Netzwerknamen und Port passende `addrinfo`-Struktur zu bekommen, mit der dann `connect()` aufgerufen werden kann. Ein passendes Codebeispiel findet ihr am Ende der Manpage von `getaddrinfo()`.

Nach dem Verbindungsaufbau liest `send` mit `getline()` zeilenweise von `stdin` und sendet den Text zeilenweise mit dem oben beschriebenen Protokoll an `merge`. Wenn das Ende der Datei erreicht wird, wird der Socket geschlossen.

Damit `send` auch wirklich wie gefordert aufsteigend sortierte Wörter und nicht unsortierte Zeilen schickt, wird die Eingabe für `send` aufbereitet. Zunächst werden alle Leerzeichen durch Zeilenumbrüche mit `tr` ersetzt. Das Ergebnis wird dann mit `sort` sortiert. Danach werden noch Duplikate mit `uniq` entfernt. `send` kann dann zum Beispiel in der Shell auf folgende Weise aufgerufen werden:

```
tr < text1.txt ' ' '\n' | sort | uniq | ./send localhost:50000
```

Aufgabe 3: Test

Testet die Anwendungen mit Texten eurer Wahl und dokumentiert eure Ergebnisse.

Hinweise

- Alle in dieser Aufgabe geforderten Programme sollen mit einem Makefile übersetzt werden können. In diesem Makefile sollen Abhängigkeiten des Build-Prozesses richtig erfasst sein. Des weiteren soll das Makefile auch eine Regel `clean` enthalten, die alle Komplilate wieder löscht.
- Achtet darauf, dass Ihr die Rückgaben aller Systemaufrufe auf mögliche Fehler überprüft. Im ungewünschten Fehlerfall soll eine erklärende Ausgabe auf `stderr` erfolgen und das Programm dann mit `exit(EXIT_FAILURE)` (siehe `man 3 exit`) beendet werden. Viele Funktionen setzen im Fehlerfall die Variable `errno` (siehe `man 3 errno`), über die der genaue Fehler identifiziert werden kann. Hilfreich für Fehlerausgaben ist die Funktion `strerror()` (siehe `man 3 strerror`) zur Formatierung von `errno`. Studiert auf jeden Fall die entsprechenden man-Pages, um zu erfahren welche Rückgabewerte Fehler kennzeichnen und welche Fehler auftreten können.
- Achtet darauf, dass Ihr allen Speicher, den Ihr mit `malloc()` alloziert habt, auch zum Programmende wieder mit `free()` freigegeben habt. Bei der Behandlung von Fehlerfällen, bei denen das Programm mit einem Fehlercode beendet wird, darf darauf verzichtet werden. Hilfreich zum Aufdecken von Speicherlecks und Speicherzugriffsfehlern ist das Tool `valgrind`.
- Die Abgabe erfolgt als Ausdruck am Ende der Übung und zusätzlich elektronisch über das Subversion Repository.
- Die Dokumentation der Aufgabenlösung ist in LaTeX anzufertigen.