Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
WS 06/07

AG Betriebssysteme FB3
Kirsten Berkenkötter, Jan Peleska

Ausgabe: 7.12.2006
Abgabe: 11.1.2007

Übungszettel 3

Hinweise

Die Abgabe erfolgt als Ausdruck am Ende der Vorlesung und als E-Mail an *kirsten@tzi.de*. **Auf jeden Fall** sollten alle C-Dateien auch in elektronischer Form (als E-Mail-Attachment) abgegeben werden. Zur vollständigen Lösung der Aufgabe gehören Programm, Test und Dokumentation (in Latex). Der Betreff der E-Mail sollte folgendes Aussehen haben:
BS1 Abgabe x Gruppe y.

Bitte immer die Namen aller Gruppenmitglieder und die Gruppennummer angeben!

Aufgabe 1: Kleine Hilfsbibliothek

Programmiert eine kleine C-Bibliothek (*udplib.c* und *udplib.h*), in der ihr Hilfsfunktionen für die Aufgaben 2 und 3 bereitstellt:

int getSharedMemory(int key, int permission, int size);

Diese Funktion liefert einen Filedescriptor für ein Shared Memory zurück. Im Fehlerfall wird das Programm mit *exit(1)* beendet. *key* ist der Schlüssselwert zur Erzeugung des Shared Memory, *permission* die zu setzenden Zugriffsrechte und *size* die Größe des Shared Memory.

void initSharedMemory(void *pointer, int size);

Diese Funktion initialisiert jedes Byte des Shared Memory mit *0*. Übergeben wird der Zeiger auf das Shared Memory, der im User-Space erzeugt wird. *size* ist die Größe des Shared Memory.

int getSemaphor(int key, int number, int permission);

Diese Funktion liefert einen Filedescriptor für ein Semaphor(array) zurück. Im Fehlerfall wird das Programm mit *exit(1)* beendet. *key* ist der Schlüsselwert zur Erzeugung des Semaphor(arrays), *number* die Anzahl der zu allokierenden Semaphore und *permission* die zu setzenden Zugriffsrechte.

void initSemaphor(int semid, int semnum, int value);

Diese Funktion initialisiert einen Semaphor. *semid* ist der Filedescriptor des Semaphorarrays, *semnum* die Nummer des zu initialisierenden Semaphors im Array und *value* de Wert, mit dem der Semaphor initialisiert werden soll.

void up(int semid, int semnum);

Diese Funktion gibt einen Semaphor frei, d.h. erhöht den Semaphor um *1*. *semid* ist der Filedescriptor des Semaphorarrays, *semnum* die Nummer des freizugebenden Semaphors im Array.

void down(int semid, int semnum);

Diese Funktion fordert einen Semaphor an, d.h. erniedrigt den Semaphor um *1*. *semid* ist der Filedescriptor des Semaphorarrays, *semnum* die Nummer des anzufordernden Semaphors im Array.

int getSocket(sPort);

Diese Funktion erzeugt mit Hilfe von *socket()*, *setsockopt()* und *bind()* einen UDP-Socket, mit dem über den Port *sPort* gesendet und empfangen werden kann. Der Rückgabewert ist der Filedescriptor des Socket. Im Fehlerfall wir das Programm mit *exit(1)* beendet.

Aufgabe 2: UDP-Host

Programmiert einen UDP-Host (*udp.c* und *udp.h*), der mit anderen Hosts über eine verbindungslose Punkt-zu-Punkt-Kommunikation Kontakt aufnehmen kann. Als Argumente sollen dem Programm der Port für den eigenen Socket sowie der Hostname und der Port für den anzusprechenden Socket übergeben werden. Bei falschen Argumenten wir das Programm mit *exit(2)* verlassen.

Wenn der eigene Port *4444* ist, und Host *auenland* auf Port *5555* angesprochen werden soll, sieht der Programmaufruf also so aus:

```
udp 4444 auenland 5555
```

Das Programm soll folgendermaßen aufgebaut werden:

Aufbau von Socket, Shared Memory und Semaphor

Mit Hilfe der Bibliothek aus Aufgabe 1 sollen ein Socket zum Senden und Empfangen, ein Shared Memory zum Speichern der gesendeten und empfangenen Daten und ein Semaphor zum Schutz des Shared Memory erzeugt werden.

Das Shared Memory besteht aus einem Puffer für die gesendeten Daten, einem Puffer für die empfangenen Daten und den jeweiligen Indizes zum Schreiben in die Puffer. Im Puffer soll Platz für mindestens zwei Datenpakete sein, die Größe des Puffers kann in *udp.h* angegeben werden. Die maximale Datenlänge pro Item beträgt *4000 Bytes*, um die zulässige Größe für UDP-Pakete nicht zu überschreiten.

Senden und Empfangen

Mit Hilfe von *fork()* soll sich das Programm in einen Kind- und einen Vaterprozess aufspalten. Der Vaterprozess übernimmt das Senden von Datenpaketen, der Kindprozess das Empfangen.

Vater Der Vaterprozess sendet nacheinander zehn Datenpakete der Form *Senden 1*, *Senden 2*, etc. Die Abstände zwischen den Datenpaketen werden mit Hilfe von *srandom()* und *random()* zufällig ermittelt.

Kind Das Kind wartet auf empfangene Datenpakete.

Behandlung der Datenpakete Die gesendeten und empfangenen Daten sollen zur Kontrolle auf dem Bildschirm ausgegeben werden. Außerdem werden sie in den Sende- bzw. Empfangspuffer geschrieben (dient der Vorbereitung von Aufgabe 3). Diese sind als Ringpuffer organisiert. Bereits gesendete Datenpakete können überschrieben werden und gehen dadurch verloren.

Aufgabe 3: UDP-Protokoll

Bei der Übertragung von Daten per UDP können Datenpakete verloren gehen. Implementiert ein kleines Protokoll, um die sichere Übertragung von Daten zu gewährleisten (*udpprotocol.c* und *udpprotocol.h*). Der Aufbau der Hosts soll dabei bis zum fork() analog zu Aufgabe 2 erfolgen, allerdings muss das Shared Memory geeignet erweitert werden, um das Protokolls verwalten zu können. Nach dem fork() laufen Vater- und Kindprozess jeweils in einer Endlosschleife.

Das Protokoll besteht aus den folgenden Nachrichten:

CON_REQ	Connection Request, Verbindungsaufbau
CON_ACK	Connection Acknowledge, Verbindungsaufbau bestätigt
CON_REJECT	Connection Reject, Verbindungsaufbau zurückgewiesen
CON_DATAx	Connection Data x, Datenpaket x gesendet
CON_DATA_ACK	Connection Data Acknowledge, Datenpaket empfangen
CON_CLOSE	Connection Close, Verbindungsende
CON_CLOSE_ACK	Connection Close Acknowledge, Verbindungsende bestätigt

Die Datenpakete brauchen nicht mit Daten gefüllt zu werden, das Senden der entsprechenden Nachrichten genügt. Eine erfolgreiche Verbindung hat also z.B. folgendes Aussehen:

```
CON_REQ
                CON_ACK
CON_DATA2
                CON_DATA_ACK
CON_DATA1
                CON_DATA_ACK
CON_CLOSE
                CON_CLOSE_ACK
```

Das Programm mit Protokoll funktioniert nun folgendermaßen:

Aufbau einer Verbindung

Es kann immer nur ein Kommunikationspartner eine Verbindung aufbauen, um Daten zu senden. Der andere Partner sendet währenddessen nur Quittungen. Nach Ende der Verbindung kann er selbst eine Verbindung aufbauen und Daten senden (wer anfängt, wird durch die zufälligen Zeiten beim Senden bestimmt, s. Aufgabe 1).

Der Vaterprozess sendet also entweder nur Nachrichten oder nur Quittungen, wobei der korrekte Ablauf beachtet werden muss. Analog dazu muss der Kindprozess Quittungen empfangen und überprüfen, bzw. Nachrichten empfangen und überprüfen.

Vater- und Kindprozess kommunizieren über das Shared Memory miteinander, um den korrekten Ablauf des Protokolls zu gewährleisten.

Erfolgreicher Ablauf

Bei einer erfolgreichen Verbindung sendet der sendende Kommunikationspartner *CON_REQ*, danach *CON_DATAmax*, *CON_DATAmax-1* bis *CON_DATA1* und schließlich *CON_CLOSE*. Der empfangende Kommunikationspartner sendet ausschließlich Quittungen *CON_ACK*, *CON_DATA_ACK* (für jedes einzelne Datenpaket) und *CON_CLOSE_ACK*. Die Anzahl der Datenpakete wird mit *random()* ermittelt.

Nicht erfolgreicher Ablauf

Kann eine Verbindung nicht angenommen werden, wird *CON_REJECT* als Quittung geschickt. Danach muss ein neuer Verbindungsaufbau vorgenommen werden. Dies ist der Fall, wenn beide Kommunikationspartner gleichzeitig eine Verbindung aufnehmen wollen.

Bei sonstigen Problemen (z.B. falsche Quittung, mehrere gleiche Nachrichten nacheinander) wird der Host wieder initialisiert und vesucht, eine neue Verbindung aufzubauen.

Ausnutzung des Puffers

Um nicht auf jede Quittung einzeln warten zu müssen, können mehrere Nachrichten nacheinander geschickt werden. Die Anzahl der noch nicht quittierten Nachrichten darf aber *Puffergroesse-1* nicht überschreiten. In diesem Fall wird ein Flag *XOFF* gesetzt. Erst wenn das Senden wieder möglich ist, wird dieses auf *XON* zurückgesetzt. Nach zehn erfolglosen Sendeversuchen bei *XOFF* wird angenommen, dass keine Quittung mehr erfolgt, ein neuer Verbindungsaufbau muss vorgenommen werden.

Beendigung

Zur korrekten Beendigung des Programms soll ein Signalhandler verwendet werden, der auf die beiden Signale *SIGINT* und *SIGTERM* hört.