Übungsaufgabe #2 Kommunikations-Basissystem für Fernaufrufe

30.04.2003

Sie haben in der Vorlesung verschiedene Kommunikationsmethoden kennengelernt. In dieser Aufgabe soll ein Basissystem erstellt werden, welches in einem ersten Schritt eine Systemabstraktion bietet mit einer einheitlichen, einfachen Schnittstelle auf verschiedene Basismechnismen des Betriebssystem (z.B. Kommunikation mit TCP oder UDP). In einem zweiten Schritt soll darauf aufbauend einige der oben erwähnten Methoden angeboten werden. Kommunikationsschnittstelle Das Kommunikationssystem ist stets auf eine objektorientierte Abstraktion mit nachfolgend beschriebenen Elementen abzubilden:

- Initialisierung: Der Konstruktur initialisiert das Kommunikationsobjekt. Er kann ein Adressobjekt als Parameter erhalten, der dann festlegt, auf welcher lokalen Adresse Daten empfangen werden können (z.B. IP-Adresse und Port für UDP). Wird keine lokale Adresse angegeben, wählt (soweit möglich) das System selbst eine. Mit der Methode `getLocalAddress` kann diese Adresse, unter der das Kommunikationssystem von aussen erreichbar ist, abgefragt werden.
- Senden von Nachrichten: Zum Senden von Nachrichten wird vom System stets eine Methode `void send(Addresse *addr, Buffer *buf)` angeboten, unabhängig von der Semantik, die dahinter steckt. Für den Fall, dass das send zurückkehrt, bevor der Puffer intern vollständig verarbeitet wurde (er darf also nicht verändert werden), ist ein Signalisierungsmechanismus notwendig, um dem Anwender mitzuteilen, dass der Puffer nicht mehr benötigt wird. Hierzu kann durch die Methode `set_send_sighand(send_sighand_t fn)` ein Signal-Handler installiert werden, der aufgerufen wird, sobald der Puffer nicht mehr benötigt wird.
- Empfangen von Nachrichten: Beim Empfangen von Nachrichten ist eine aktive (wartend) und eine passive (signalisierend) Übergabe möglich. Eine Methode `receive(Buffer *buf)` ist für die aktive Variante bereitzustellen. Diese blockiert, bis eine Nachricht empfangsbereit ist. In der passiven Variante kann durch `set_receive_sighand(Buffer *buf, recv_sighand_t fn)` dem Kommunikationssystem einen Empfangspuffer bereitstellen. Der Signalhandler wird mit Referenz auf den Puffer aufgerufen, sobald Daten empfangen worden sind.

Hierbei gelte:

```
typedef void(*send_signal_t)(Buffer *buf);
typedef void(*recv_signal_t)(Address *sender, Buffer *buf);
class Buffer { public: char *buffer; int len; }
```

a) Implementierung Teil a: Unterste Systemabstraktion:

In dieser Teilaufgaben sollen objektorientierte Abstraktionen für ein einfaches Kommunikationssystem, eine einfache Semaphore und eine Thread-Verwaltung erstellt werden. Das Kommunikationssystem soll jeweils folgende Funktionalität bereitstellen, jeweils in einer Implementierung für TCP und für UDP:

- Als Adresse ist eine Klasse `InetAddress` zu verwenden, die eine `struct sockaddr_in addr` enthält. Eine Initialisierung über Hostname und Portnummer ist vorzusehen.
- Der Anwender dieser einfachen, untersten Schicht muss damit rechnen, dass der Sendepuffer nach Rückkehr aus dem `send`-Aufruf vom System noch benötigt wird. Ein Signalisierungsverfahren wie oben beschrieben ist daher zwingend notwendig. (Für TCP und UDP müsste das nicht so sein, aber bei Abbildung auf andere Betriebssystemmechanismen kann dies erforderlich sein!). Die unterste Schicht kann sich darauf verlassen, dass send nicht nebenläufig aufgerufen wird.
- Zum Empfangen wird das beschriebene Signalisierungsverfahren verwendet.

Ein mögliches Interface könnte also wie folgt aussehen:

```
class CommunicationSystem {
 public:
    CommunicationSystem(Address *addr);
    Address *getMyAddress();
    void send(Address *dest, Buffer *message);
    void set_send_signaler(send_signal_t fn);
    void set_receive_signaler(Buffer *buffer, recv_signal_t f);
}
```


© 2003 Universität Erlangen Nürnberg, Institut für Informatik 4

Als zweites ist eine binäre Semaphore bereitzustellen mit folgender Schnittstelle:

```
class Semaphor{
 public:
  Semaphor();
  void P(void); void V(void);
};
```

Als drittes ist eine einfache Thread-Verwaltung bereitzustellen, die intern direkt auf entsprechende Funktionen der pthread-Bibliothek abgebildet werden können. Der Konstruktor von Thread erzeugt einen neuen Thread, in welchem die Methode run der übergebenen Klasse (welche Runnable implementieren muss) ausführt. Falls in den folgenden Aufgaben weitere Thread-Methoden benötigt werden, so kann diese Klasse erweitert werden. Im Hinblick auf leichte Portierbarkeit ist die Schnittstelle dabei aber auf eine Minimallösung zu beschränken.

```
class Runnable { virtual void run() = 0; }
class Thread {
 private: ...
 public: Thread(Runnable run); void join();
}
```

b) Implementierung Teil b:

Es sollen nun drei verschiedene Kommunikationsvarianten auf der Sendeseite implementiert werden, die auf der Schicht aus Teil a aufbauen.

- Kommunikationsklasse mit synchronem send:
  Hier soll eine Methode `send` implementiert werden, die auf unterer Ebene ein send aus Teil a verwendet. Zusätzlich soll sie mit parallelem Aufrufen aus mehreren Threads zurechtkommen (MT-safe). Synchron heisst, dass sich der `send`-Aufruf blockiert, bis die untere Schicht das abgeschlossene Versenden signalisiert.
- Kommunikationsklasse mit asynchronem, puffernden send
  Hier sieht die Kommunikationsklasse intern eine feste Anzahl an Puffern vor. Bei Aufruf von send werden die Daten in den internen Puffer kopiert, und send kehrt sofort zum Aufrufer zurück. Solange keine freien Puffer verfügbar sind, blockiert der `send`-Aufruf. Der Signalisierungsmechanismus der unteren Schicht wird verwendet, um Puffer freizugeben.
- Kommunikationsklasse mit asynchronem, nicht-blockierenden send
  Das interne Blockieren bei asynchronem send lässt sich vermeiden, wenn der Pufferspeicher vom Benutzer selbst verwaltet wird. Hier soll also eine Lösung ähnlich der vorangegangenen erstellt werden, bei der aber kein interner Puffer vorgehalten wird, sondern die bei send übergebenen Puffer selbst hierzu verwendet werden. Geeignete Datenstrukturen sind hierzu zu überlegen. Wie bei der Basis-Kommunikation muss hier dem Aufrufer signalisiert werden, wenn der Puffer nicht mehr benötigt wird.

c) Implementierung Teil c:

Die Implementierung von Teil (a) ist auf Microsoft Windows / Visual Studio zu portieren. Mit der Portierung sollte Teil (b) unverändert zusammenarbeiten.
Die Implementierung dieses Teils wird nicht zwingend bis zum festgelegten Abgabetermin erwartet. Spätere Teilaufgaben werden aber darauf aufbauen, eine Bearbeitung ist daher unerlässlich und sollte zeitnah erfolgen!

Alle benötigten Dateien sind im Verzeichnis /proj/i4vs/*loginname*/aufgabe2/ abzulegen.

<u>Abgabe</u>: bis 13.05.2003 12:00 Uhr

© 2003 Universität Erlangen Nürnberg, Institut für Informatik 4