Übungsaufgabe #2: Kommunikations-Basissystem: BS-Abstraktionen

10.05.2004

Sie haben in der Vorlesung verschiedene Kommunikationsmethoden kennengelernt (z.B. synchron; asynchron; blockierend; nicht blockierend). In der Aufgabe 3 (nächste Woche) werden mehrere dieser Methoden betriebssystemunabhängig realisiert. Als Vorarbeit dazu wird in dieser Aufgabe ein Basissystem erstellt, das eine Systemabstraktion mit einer einheitlichen, einfachen Schnittstelle auf verschiedene Basismechnismen des Betriebssystem bietet (Nachrichtenkommunikation mit z.B. TCP oder UDP; Semaphore zur Synchronisation; einfache Thread-Verwaltung). Dieses Basissystem ist sowohl für Unix (Linux/Solaris) als auch für Windows zu implementieren.

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(Address *addr, Buffer *buf)` angeboten, unabhängig von der Semantik, die dahinter steckt. Falls 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_signal_t  sigobj)` ein Signal-Handler installiert werden, der aufgerufen wird, sobald der Puffer nicht mehr benötigt wird.

```
class send_signal_t {
  public:
        virtual void releaseBuffer(Buffer *buf) = 0;
};
class Buffer { public: char *buffer; int len; };
```

- *Empfangen von Nachrichten:* Beim Empfangen von Nachrichten ist eine aktive (wartende) und eine passive (signalisierende) 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 dem Kommunikationssystem ein Empfangspuffer bereitgestelt werden durch `set_receive_sighand(Buffer *buf, recv_signal_t sigobj)`. Der Signalhandler wird mit Referenz auf den Puffer aufgerufen, sobald Daten empfangen worden sind.

```
class recv_signal_t {
  public:
        virtual void receiveBuffer(Address sender, Buffer *buf) = 0;
};
```

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);
    virtual Address getMyAddress();
    virtual void send(Address dest, Buffer *message);
    virtual void set_send_signaler(send_signal_t *sigobj);
    virtual void receive(Buffer *buffer);
    virtual void set_receive_signaler(Buffer *buffer, recv_signal_t *sigobj);
};
```

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 (Unix) bzw. der entsprechenden Windows-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();
};
```

a) Implementiere die beschriebene Funktionalität mit Unix. Es sollte auf Portabilität geachtet werden. (z.B. sowohl mit PC/Linux als auch mit Sun/Solaris testen!)

b) Implementiere die beschriebene Funktionalität mit Microsoft Windows / Visual Studio

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

Abgabe: bis 21.05.2004 12:00 Uhr