Kapitel 18
Programm- und Prozesssteuerung

18.1 Allgemeines

Die bislang diskutierten Anwendungsfälle setzen stillschweigend voraus, dass sich alles auf einer Maschine abspielt und sich eine Anwendung auch nur um eine Sache kümmert. In der Praxis ist das oft nicht der Fall.

Die Abweichung von der Norm beginnt bereits beim Betriebssystem, dass mehrere Aufgaben „gleichzeitig" durchführen muss. Eine gewissen Gleichzeitigkeit wird zwar heute bereits dadurch hergestellt, dass viele Maschinen über mehrere Prozessoren verfügen, aber eine Anwendung läuft meist komplett auf einem Prozessor ab und wird vom Betriebssystem oft bei der Arbeit unterbrochen, um anderen Anwendungen Gelegenheit zu geben, ihren Teil der Arbeit zu verrichten. Bei diesen Unterbrechungen müssen alle Programmzustände bis zu einer Fortsetzung gesichert werden, außerdem muss das Betriebssystem dafür sorgen, dass Daten einer Anwendung nicht durch eine andere zerstört werden und verschiedene Hardwareeinheiten reibungslos miteinander arbeiten. Dahinter stecken sehr komplexe Philosophien und (meist nicht in C++ realisierte) Programmiertechniken, die Bücher mit ähnlicher Dicke wie dieses erfordern. Wir vertiefen dieses Thema daher an dieser Stelle nicht.

Etwas mehr zum Thema passen Serverdienste in Netzwerken. Server wickeln zwar Einzelaufgaben wie die Übertragung einer Datei oder die Abfrage einer Datenbanktabelle für einen Client ab, haben es aber oft mit sehr vielen Clientsystemen gleichzeitig zu tun, die individuell bedient werden müssen. Man kann sich zwar nun interne Datenstrukturen überlegen, die die unterschiedlichen Clientzustände verwalten, aber das führt bereits bei einfachen Anwendungen zu recht komplexen und damit fehleranfälligen Programmierungen. Man wählt deshalb meist einen anderen Weg.

Die Betriebssysteme bieten Methoden an, um Prozesse zu klonen. Wird eine Anwendung in den Arbeitsspeicher geladen, so sorgt das Betriebssystem für die Bereitstellung des notwendigen Speicherplatzes sowie ein von der absoluten Lage im Arbeitsspeicher unabhängiges Adressierungsschema von Speicherstellen. Die Anwendung wird so zu einem „Prozess", der mit anderen wie angedeutet verwaltet und ausgeführt wird. „Klonen" bedeutet, dass das Betriebssystem den Code ein

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_18, © Springer-Verlag Berlin Heidelberg 2010

weiteres Mal im Speicher als 1:1-Kopie des laufenden einschließlich aller Daten platziert und ausführt. Unter Linux erfolgt dies durch die Methode

```
int pid = fork();
```

wobei an der Antwort zu erkennen ist, welche Kopie der Eltern- und welche der Kindprozess ist. Während der erste nun weiter darauf wartet, dass neue Clientsysteme an den Server binden, arbeitet der zweite selektive die Bedürfnisse eines Clients ab und beendet sich dann. Beide Prozesse verfügen nach Ausführen des Klonbefehls über den gleichen Datenbestand, so dass sich auch hier eine weitere technische Diskussion erübrigt.

Sehr komplexe Aufgaben werden u.U. von mehreren verschiedenen Anwendungen ausgeführt. Als Beispiel denke der Leser an die Darstellung von aus dem Internet mittels eines Browsers abgerufenen Dateien mit Textverarbeitungsprogrammen, die automatisch vom Browser aktiviert werden. Die Ausführung erfolgt über Betriebssystemmethoden wie

```
int pid = exec(char const* path, char const** args,..),
```

die das angegebene Programm mit den ebenfalls angegebenen Argumenten starten. Die Betriebsparameter, mit denen das gestartete Programm arbeiten soll, werden in den Argumenten der `main(..)` - Funktion übergeben.[1] Abgesehen von ausführungstechnischen Details, ob etwa der Elternprozess den Kindprozess in die Freiheit entlässt oder ihn mit Hilfe weiterer Betriebssystemmethoden nach Bedarf mit Gewalt beendet, gibt es auch hier wenig Technisches zu diskutieren.

> **Aufgabe.** ... was nicht heißt, dass die Realisierung so problemlos ist, wie sich das anhört – aber das gilt sicher für so manches in diesem Buch. Stellen Sie die von dem von Ihnen verwendeten Betriebssystem für diese Zwecke bereitgehaltenen Methoden zusammen und machen Sie einige Versuche, Prozesse zu erzeugen und zu beenden. Achten Sie darauf, dass beendete Kindprozesse dem Elternprozess signalisiert und dies oft auch quittiert werden muss, da ansonsten so genannte Zombielisten entstehen. Das Betriebssystem für die Prozessnummern der bereits beendeten Prozesse bis zur Quittierung weiter, was Ressourcen für die bereits speichertechnisch toten Prozesse verbraucht.

Die strikte Trennung von Daten verschiedener Anwendungszweige funktioniert allerdings nicht in allen Anwendungen. Manche nebeneinander abzuwickelnde Aufgaben müssen auf den gleichen Datenbestand zurückgreifen, was durch so genannte „Threads“ erledigt wird, also Prozesse, die innerhalb des gleichen Speicherbereichs arbeiten, das führt zu weiteren Problemen, die wir unten ausführlich diskutieren.

Andere Aufgaben lassen sich zwar durch eine Anwendung erledigen, besitzen aber Teile, die unabhängig voneinander erledigt werden können. Wird die Arbeit

[1] Reicht das nicht, weil sehr komplexe Daten übergeben werden müssen, kann man auf Dateischnittstellen zurückgreifen.

zeitlich sehr aufwändig, kann versucht werden, diese Teile auf verschiedenen Maschinen parallel laufen zu lassen. Aufgrund der Anwendungsbezogenheit ist das zwar nur relativ schwer systematisierbar, aber einige Worte werden wir das als Abschluss dieses Buches weiter unten verlieren.

18.2 Threads

18.2.1 Allgemeines

Threads lassen sich, wie oben angemerkt, als Prozesse in Prozessen betrachten. Damit ist auch die Programmierumgebung in die Gestaltung eingebunden. Die folgenden Ausführungen betreffen vorzugsweise das UNIX/POSIX-Schema. Sofern Sie andere Umgebungen verwenden, können mehr oder weniger große Abweichungen von den Beschreibungen auftreten.

Bei Threads ist zu beachten, dass sich die Prozesse die Ressourcen teilen. Im Klartext bedeutet dies, dass alle im sichtbaren Bereich der Threadfunktionen befindlichen Funktionen, Objektmodule und globale Variable in nicht vorhersehbarer Weise von den Threads angesprochen werden können. Aufrufe von Funktionen, die nicht auf statische Objekte zurückgreifen, sind unkritisch, da jeder Thread über einen eigenen Stack verfügt.[2] Statische und globale Variable können aber durch einen Thread verändert werden, bevor ein anderer Thread seine Arbeit abgeschlossen hat, und das kann fatale Auswirkungen haben. Stellen Sie sich eine Steuerung vor, in der ein Thread prüft, ob ein Ventil geöffnet ist, um anschließend eine Pumpe einzuschalten. Bevor er dazu kommt, wird er von einem zweiten Thread unterbrochen, der feststellt, dass die Pumpe nicht läuft, und das offene Ventil schließt, worauf der erste Thread nach Rückkehr die Pumpe mit voller Leistung auf das geschlossene Ventil loslässt.

Bei einem Zugriff auf gemeinsam genutzte Ressourcen müssen die Threads somit synchronisiert werden. Im einfachsten Fall bedeutet dies, dass ein Thread mit einem Zugriff warten muss, bis ein anderer seine Arbeit beendet hat, in komplizierteren Fällen kann es erforderlich sein, dass ein anderer Thread seine Arbeit erst noch tun muss, bevor etwas in Angriff genommen wird.

ACHTUNG ! Das betrifft auch Bibliotheks- und Systemfunktionen sowie den Compiler. Da man hier in der Regel nur über die Headerdateien verfügt, weiß man nicht, wie die Arbeit im Detail durchgeführt wird und wie sich eine Unterbrechung durch einen anderen Thread auswirkt. Bei Bibliotheken ist auf eine „thread-safe“-Version zu achten (*ansonsten muss man das thread-safe-Verhalten*

---

[2] Der Arbeitsspeicher wird normalerweise in Programm-, Daten-, Freispeicher- und Stackbereich eingeteilt, wobei die letzten beiden (variablen) Bereiche jeweils von einem Ende des zur Verfügung stehenden Speichers aufeinander zulaufen. Man kann sich leicht vorstellen, dass die Verwaltung mehrerer unabhängiger Stackbereiche eine anspruchsvolle Aufgabe darstellt, die einigen Platz in Lehrbüchern über Betriebssysteme einnimmt.

selbst herstellen), beim Übersetzen des Programms ist möglicherweise auf eine Optimierung zu verzichten, da hierdurch wesentliche Teile der Threadsteuerung entfernt werden können. Konsultieren Sie also in jedem Fall vor den folgenden Versuchen die Dokumentation Ihres Systems, um unliebsamen Überraschungen, an denen Ihre Kodes keinerlei Schuld tragen, aus dem Weg zu gehen!

18.2.2 Erzeugen und Kontrollieren

Beginnen wir mit der Erzeugung eines Threads mittels der Funktion `pthread_create`.

```
int pthread_create(pthread_t* thread,
              const pthread_attr_t* attr,
              void *(func)(void*),
              void* arg);
```

`pthread_t` ist eine Datenstruktur, die den erzeugten Thread identifiziert. Wenn der erzeugende Prozess eine Kontrolle über die Threads ausüben will, muss er für jeden erzeugten Thread eine Variable dieses Typs vorhalten. Was sich dahinter verbirgt, soll hier nicht weiter interessieren. Wenn die Abwicklung der Threads effektiv erfolgen soll, sind tiefe Rückgriffe in die Prozessverwaltung des Betriebssystems notwendig, und bei Interesse an diesem Thema sollten Sie ein Lehrbuch über Betriebssystembau zu Rate ziehen.

`pthread_attr_t` erlaubt es, dem Thread bestimmte Ausführungseigenschaften (*Priorität*) zuzuordnen. In der Regel wird man dies nicht benötigen und kann eine Null als Aufrufparameter übergeben.

Die gestartete Funktion ist vom Typ `void* f(void*)`. Der `void`-Parameter wird als vierter Aufrufparameter übergeben, so dass beliebige Daten in den Thread transportierbar sind. Es ist jedoch darauf zu achten, dass es sich hierbei um einen Zeiger handelt. Ändert man den Wert, der hinter dem Zeiger steht, so ist dies auch in der Threadfunktion sichtbar, und man hat ein einfaches Modell für einen individuellen Nachrichtentransport. Wird der Speicherbereich des Zeigers vorzeitig freigegeben, so besitzt der Thread nun einen Zeiger auf einen ungültigen Speicherbereich und löst beim nächsten Zugriff eine Speicherverletzung aus.

Der Gesamtaufruf für einen Thread ist mithin

```
void* f(void*) { cout << "Hallo Welt" << endl;}

int main(){
    pthread_t p;
    pthread_create(&p,0,f,0);
    ...
```

Der gestartete Thread läuft nun parallel zum übrigen Geschehen so lange, bis die Thread-Funktion wieder verlassen wird. Alle Threadfunktionen können auf globale Daten uneingeschränkt zugreifen; lokale Daten sowohl der Kind- als auch der Elternthreads sind wie üblich lokal.

Da mit dem Ende des main-Programmteils auch das Ende der gesamten Anwendung gekommen ist, sollte zumindest so lange gewartet werden, bis der Thread beendet ist. Für diese Synchronisationsaufgabe sind zwei Funktionen vorhanden:

```
void pthread_exit(void *retval);
int pthread_join(pthread_t thread,
                 void **value_ptr);
```

Die erste Funktion wird vom Thread am Ausführungsende aufgerufen, anstatt einfach den Code enden zu lassen oder eine `return`-Anweisung einzufügen (*der Aufruf kehrt nicht mehr zurück*). Über die Zeigervariable kann eine Nachricht an den erzeugenden Prozess zurückgegeben werden, wobei es sich hierbei nicht um einen Zeiger auf eine lokale Variable handeln darf, da diese mit beenden des Threads nicht mehr definiert ist. Bei Aufruf der zweiten Funktion im erzeugenden Thread wird die Ausführung unterbrochen, bis der angegebene Thread beendet ist. Der in `pthread_exit` übergebene Zeigerwert wird an der angegebenen Adresse abgespeichert. In einem kleinen Beispiel sieht das folgendermaßen aus:

```
void* f(void* ptr) {
    int* pi = (int*) ptr;  int* pj;
    pj=new int; *pj=*pi+10;
    pthread_exit((void*)pj);
}//end function

int main(){
    pthread_t p;
    int value=10;
    int* pi;
    pthread_create(&p,0,f,(void*)&value);
    pthread_join(p,(void**)&pi);
    cout << *pi << " " << value << endl;
    delete pi;
```

Durch ein einfaches ThreadManager-Objekt lässt sich das Synchronisieren absichern. Die Thread-Identifier werden in einem Container gesammelt. Im Destruktor wird `pthread_join` für alle noch vorhandenen Threads aufgerufen, so dass der kontrollierende Thread nicht vor seinen Kindern beendet werden kann. `last_id` gibt den Thread-Identifier des letzten erzeugten Threads zurück, so dass der Elternthread das Beenden der Kinder auch selbst kontrollieren kann.

```
class ThreadManager {
public:
    typedef void*(*func)(void*);

    ThreadManager();
    ~ThreadManager();
```

```
    template <typename f>
    bool create(f fu){ return Create((func)fu,0); }

    template <typename f, typename arg>
    bool create(f fu, arg& v)
      { return Create((func)fu,(void*)&v); }

    pthread_t last_id() const;

    bool sync(pthread_t const& pt);
    template <typename arg>
    bool sync(pthread_t const& pt, arg& v)
        { return Sync(pt,(void**)&v);}
private:
    bool Create(func fu, void* v);
    bool Sync(pthread_t const& pt, void** v);

    ThreadManager(ThreadManager&);
    deque<pthread_t> threads;
};//end class
```

Aufgabe. Implementieren Sie die Funktionen der Klasse `ThreadManager`.

18.2.3 Exklusive Programmteile

Wie in der Einleitung beschrieben, muss die Arbeit von Threads, die auf gemeinsame Ressourcen zugreifen, synchronisiert werden. Sobald ein Thread einen solchen Programmbereich erreicht, muss verhindert werden, dass andere Threads vor Abschluss der Arbeit ebenfalls auf diesen Bereich zugreifen.

Man könnte nun versuchen, dies mit logischen Variablen zu regeln, indem ein Thread bei Erreichen des kritischen Bereiches prüft, ob schon jemand dort arbeitet, und im Negativfall die Sperrvariable bedient. Man stellt jedoch fest:

- Auch die Bedienung der Sperrvariablen ist ein kritischer Bereich, und es lässt sich nicht oder nur mit großen Aufwand verhindern, dass es auch hierbei zu Konflikten kommt.
- Bei einer derartigen Sperre kann der Thread nur in einer Endlosschleife darauf warten, dass der Bereich freigegeben wird. Wartende Threads verbauchen dann aber Rechenzeit, ohne tatsächlich etwas zu tun.

Eine effektive Sperre lässt sich also nur mit Hardwarehilfen (*spezielle Maschineninstruktionen, die garantiert nicht unterbrochen werden können*) und mit Rückgriff auf das Betriebssystem, dass einem wartenden Thread bis zur Freigabe der Ressource keine Rechenzeit mehr zuteilt, realisieren. Für das Anwendungsprogramm stellt die Bibliothek so genannte Mutexe (*von mutual exclusion*) zur Verfügung.

```
// Initialisierung
pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER;

// Beginn des kritischen Bereiches
pthread_mutex_lock(&mlock);
....
// Ende des kritischen Bereiches
pthread_mutex_unlock(&mlock);
```

In der Anwendung können beliebig viele Mutexe deklariert und verwendet werden, so dass unterschiedliche kritische Bereiche auch einzeln kontrolliert werden können. Allerdings ist Aufmerksamkeit geboten:

```
Thread 1                    Thread 2
-------------------------------------------------
mutex_lock(&l1)             mutex_lock(&l2)
...                         ...
mutex_lock(&l2)             mutex_lock(&l1)
...                         ...
```

Die Threads in diesem Beispiel belegen mehrere Mutexe, was zunächst kein Problem ist, aber in umgekehrter Reihenfolge. Setzt Thread 1 den Mutex `l1`, Thread 2 gleichzeitig `l2`, so hängen beide bei dem Versuch, den zweiten Mutex für sich zu reservieren, fest. Eine solche Situation heißt Deadlock, und da es sich um einen grundsätzlichen Programmierfehler handelt, existieren auch keine Abhilfestrategien.

Für die Nutzung von Bibliotheksfunktionen, bei denen man nicht sicher ist, ob sie auch Thread-Safe implementiert sind, hilft ein kleiner Wrap-Around:

```
void SystemMutex(bool lock);

template <typename f> void tsafe(f func){
    SystemMutex(true);
    func();
    SystemMutex(false);
}//end function

template <typename f, typename arg>
void tsafe(f func, arg arg1){
    SystemMutex(true);
    func(arg1);
    SystemMutex(false);
}//end function
...
```

Modul mit dem Mutex

```
static pthread_
     mutex_t mlock = PTHREAD_MUTEX_INITIALIZER;

void SystemMutex(bool lock){
    if(lock)    pthread_mutex_lock(&mlock);
    else        pthread_mutex_unlock(&mlock);
};//end class
```

Vereinigt man alle Attribute und Methoden eines kritischen Bereiches in einer Klasse und sorgt dafür, dass jeweils nur ein Thread eine Methode der Klasse ausführen kann, so nennt man eine solche Klasse einen „Monitor“. Im Prinzip wird zu Beginn jeder Methode `pthread_mutex_lock` aufgerufen, bei Verlassen der Methode `pthread_mutex_unlock`. Leider existiert in C++ keine Möglichkeit, solche Funktionsaufrufe automatisch einzubauen. Eine Basisklasse, von der alle Monitore erben müssen, und ein Makro für die Implementation vereinfachen das Problem jedoch erheblich:

```
#define FUNCTION(RETURN,CLASS,FUNC)   \
    RETURN CLASS::FUNC { MUTEX __mutex(*this);

class Monitor {
public:
    Monitor();
    virtual ~Monitor();

protected:
    struct MUTEX {
        MUTEX(Monitor const& m);
        ~MUTEX();
        pthread_mutex_t& mutex;
    };//end struct

    mutable pthread_mutex_t mutex;
};//end class

Monitor::Monitor(){
    mutex = PTHREAD_MUTEX_INITIALIZER;
}//end function

Monitor::~Monitor(){}

Monitor::MUTEX::MUTEX(Monitor const& m): mutex(m.mutex) {
    pthread_mutex_lock(&mutex);
}//end function

Monitor::MUTEX::~MUTEX(){
    pthread_mutex_unlock(&mutex);
}//end function
```

Eine Funktion einer Monitorklasse

```
class A {
public:
  resultType func(objType& ob);
...
```

wird mittels des Makroaufrufs

```
FUNCTION(resultType,A,func(objType& obj))
   ...
```

implementiert. Das Makro sorgt für die Deklaration einer Variable `__mutex`, die im Kon- und Destruktor den klasseneigenen Mutex bedient. Da die Variable bei Verlassen der Funktion abgebaut wird, und zwar unabhängig von der Art, wie die Funktion verlassen wird, wird der Mutex am Ende der Arbeit wieder freigegeben.

Auf zwei Punkte ist bei der Verwendung von Monitoren unbedingt zu achten:

- Das Monitorobjekt ist als globale Variable, auf das alle Threads Zugriff haben, zu implementieren.

  Es ist keinesfalls innerhalb eines Threads ein Monitor zu instanziieren, da jeder Monitor über einen eigenen Mutex verfügt. Ein Verriegeln gegen andere Threads ist dann nicht möglich.
- Aus einer Monitorfunktion darf keine andere Monitorfunktion aufgerufen werden, die mittels des Makros implementiert sind, da sich der Thread sonst selbst blockiert.

  Ein Aufruf von weiteren, ohne Verwendung des Makros implementierten Funktionen ist möglich.

  Alle `public`-Funktionen sind mittels des Makros zu implementieren, alle intern verwendeten Methoden sind im `protected/private`-Bereich ohne Verwendung des Makros anzulegen.[3]

18.2.4 Synchronisation von Threads

Die Arbeit von Threads ist nicht nur in kritischen Bereichen exklusiv zu gestalten, Threads müssen unter Umständen auch eineinader zuarbeiten können. Wenn ein Thread feststellt, dass eine bestimmte Voraussetzung noch nicht eingetreten ist, muss er in einen Wartezustand gehen, bis ein anderer Thread die Voraussetzung hergestellt hat. Durch die Verwendung von Mutexes wäre das zwar realisierbar, aber wiederum mit einem Laufzeitverlust verbunden, da keine Möglichkeit besteht, einen Thread bis zum Vorliegen der Voraussetzung anzuhalten.

---

[3] In beiden Bereichen verwendbare Funktionsabläufe sind durch eine makroimplementierte Aufruffunktion im `public`-Bereich für die im `protected`-Bereich angelegte eigentliche Funktionsdurchführung zu realisieren

Dies wird mit Hilfe von Bedingungssignalen erreicht. Der abhängige Thread gibt dem Betriebssystem bekannt, dass er auf ein bestimmtes Signal wartet, und geht in den Ruhezustand. Das Betriebssystem wiederum reaktiviert den Thread, sobald von einem anderen das Signal gesendet wird. Für die Ablaufsteuerung wird ein Mutex und ein Signal benötigt, die zunächst wieder beide global zu deklarieren sind, damit die beteiligten Threads darauf zugreifen können.

```
pthread_mutex_t lock= PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  sig = PTHREAD_COND_INITIALIZER;
```

Thread 1 aktiviert nun zunächst den Mutex, da die Prüfung der Voraussetzung auf jeden Fall einen kritischen Bereich betrifft. Ist die Voraussetzung nicht erfüllt, wird das Signal aktiviert:

```
pthread_mutex_lock(&lock);
...
if(...) pthread_cond_wait(&sig,&lock);
...
pthread_mutex_unlock(&cond_lock);
```

Thread 2 geht ähnlich vor, nur dass er ein Signal sendet, sobald er die Bedingung erfüllt hat.

```
pthread_mutex_lock(&lock);
...
pthread_cond_signal(&sig);
...
pthread_mutex_unlock(&lock);
```

Der Ablauf ist nun folgendermaßen:

- Thread 1 setzt den Mutex und verhindert hierdurch, dass ein anderer Thread in den kritischen Bereich gelangt. Kann er alles bearbeiten, setzt der den Mutex nach Abschluss der Arbeit zurück.
- Ist die Bedingung nicht erfüllt, wird durch `pthread_cond_wait` der Mutex zurückgesetzt und der Thread gestoppt. Thread 2 kann nun ausgeführt werden.
- Thread 2 arbeitet seinen Code vollständig ab, wobei er das Signal aussendet.
- Nach Freigabe des Mutex durch Thread 2 wird Thread 1 hinter der Wait-Funktion fortgesetzt. Der Mutex wird dabei wieder für Thread 1 aktiviert und später vom Thread freigegeben.

Wird die Bedingung nicht mehr benötigt, so werden weitere Aufrufe von `pthread_cond_signal` ignoriert. Signale werden also nur dann ausgewertet, wenn tatsächlich ein anderer Thread darauf wartet. Das ist beim Programmentwurf zu berücksichtigen. Gelangt Thread 2 zuerst in Besitz des Mutex, verhindert also die Ausführung von Thread 1, so muss hierdurch die Bedingung für die Ausführung von Thread 1 geschaffen werden, so dass letzterer nach Erhalt des Mutex nicht mehr auf das Signal warten muss (*das inzwischen ins Leere gesendet wurde*).

Anstelle der Funktion `pthread_cond_signal` ist `pthread_cond_broadcast` zu verwenden, wenn mehr als ein Thread auf das Signal wartet. Das Betriebssystem aktiviert dann alle wartenden Threads nacheinander (*die erste Funktion aktiert nur jeweils einen Thread, bei mehreren also zufällig einen der wartenden*).

Da die Bedienung der Mutex- und der Bedingungsvariablen an mehreren Stellen erfolgt und der Ablauf dazwischen individuell von der Anwendung abhängt, ist organisatorisch durch spezielle Verwaltungsobjekte kaum etwas zu gewinnen. In einigen Anwendungen ist eine strikte Synchronisation aber gar nicht notwendig; vielmehr sollen die Threads im Mittel das gleiche Arbeitspensum bewältigen. Dies lässt sich durch „Arbeitstakte“ kontrollieren, die an strategischen Stellen im Anwendungsprogramm ausgelöst werden. Überschreitet ein Thread einen vorgegebenen Vorsprung von Arbeitstakten vor seinen Kollegen, ist er zu stoppen, bis die anderen den Vorsprung aufgeholt haben, was durch die ihnen nunmehr zukommende zusätzliche Arbeitszeit beschleunigt wird. Für diese Aufgabe definieren wir ein Laststeuerungsobjekt, das individuell in jedem Thread erzeugt und angestoßen wird:

```
class Laststeuerung {
public:
    Laststeuerung(int gr, int ahead);
    ~Laststeuerung();

    void signal() const;
    int get_id() const { return idnr; }

private:
    Laststeuerung();
    Laststeuerung(Laststeuerung const&);

    int group;
    int vorl;
    int idnr;
};//end class
```

Um verschiedene Gruppenarbeiten zu koordinieren, wird jeder Thread einer Gruppe zugeordnet. Jede Gruppe wiederum kann mehrere Thread verwaltung. Hierzu verfügt sie über einen Gruppenmutex und für jeden Thread über eine Bedingungsvariable. Weitere Merker sichern die Taktzähler sowie den Haltevermerk, mit dem die Signale zum Fortsetzen der Arbeit erzeugt werden.

```
struct LastGroup {
    int group;
    pthread_mutex_t mutex;
    vector<int>             id;
    vector<pthread_cond_t>  cond;
    vector<int>             count;
    vector<bool>            halted;
```

```
    LastGroup(int gr): group(gr) {
        mutex=PTHREAD_MUTEX_INITIALIZER;
    }//end constructor
};//end struct

static vector<LastGroup> lgroups;
```

Mit Erzeugen einer Instanz von `Laststeuerung` wird bei Bedarf eine Gruppe erzeugt bzw. in einer Gruppe ein neuer Eintrag. Bei Verschwinden der Instanz werden die Daten des Threads aus der Gruppe gelöscht, nach Verschwinden aller Threads einer Gruppe schließlich auch die Gruppe. Die Containergrößen sind somit variabel, was bei der Implementation zu berücksichtigen ist.

Zur Bearbeitung eines Taktsignals muss zunächst die Gruppe gesucht und deren Mutex gesperrt werden. Da hierbei möglicherweise ein Kontrollobjekt eines anderen Thread erzeugt wird oder verschwindet, ist die Suchaktion durch einen globalen Mutex zu schützen, der nach Setzen des speziellen Gruppenmutex wieder zurückgesetzt wird.

```
void Laststeuerung::signal() const {
    pthread_mutex_lock(&lgroup_mutex);
    g_iterator it=find_group(group);
    pthread_mutex_lock(&it->mutex);
    pthread_mutex_unlock(&lgroup_mutex);
```

Nach setzen des speziellen Mutex können Threads anderer Gruppen weiterarbeiten. Das Erzeugen und Zerstören von Instanzen muss aber alle Mutexe berücksichtigen und darf nicht anlaufen, so lange ein Thread in der `signal`-Funktion arbeitet. Nach Blockieren der anderen Threads wird die Taktzählvariable bedient und auf Überlauf kontrolliert.

```
int ofs, mm, i;
for(ofs=0;ofs<it->id.size();ofs++)
    if(it->id[ofs]==idnr) break;

it->count[ofs]++;
if(it->count[ofs]<0)
   fill(it->count.begin(),it->count.end(),0);
```

Blockierte Threads können freigegeben werden, sobald ihr Taktzähler das Schlusslicht der Kontrolle bildet.

```
mm=*min(it->count.begin(),it->count.end());
for(i=0;i<it->halted.size();i++){
    if(it->halted[i] && it->count[i]==mm){
        it->halted[i]=false;
        pthread_cond_signal(&it->cond[i]);
    }//endif
}//endfor
```

Abschließend wird geprüft, ob der Thread unterbrochen werden muss.

```
    if(it->count[ofs]-mm>=vorl){
        it->halted[ofs]=true;
        pthread_cond_wait(&it->cond[ofs],
                          &it->mutex);
    }//end function

    pthread_mutex_unlock(&it->mutex);
}//emnd function
```

Aufgabe. Implementieren Sie den Konstruktor und den Destruktor des Lastkontrolle-Objekts anhand der Vorgaben.

Die Lastkontrolle sorgt nur für ein ungefähres Gleichtakten der Threads, berücksichtigt jedoch nicht die Arbeit in kritischen Bereichen. Diese muss getrennt auf die beschriebenen Art realisiert werden.

18.3 Kommunikation zwischen Prozessen

18.3.1 Sockets

Wenn die Kommunikation zwischen verschiedenen Prozessen in Echtzeit erfolgen muss, bedient man sich zweckmäßigerweise der gleichen Technik, die auch in Netzwerken verwendet wird. Formal hat dabei ein Partner die Serverrolle, d.h. er wartet darauf, dass sich jemand an ihn mit einem Dienstwunsch wendet, der andere die Clientrolle, d.h. er wendet sich an einen Server mit einer Anfrage.

Ohne jetzt zu weit in die Details eindringen zu wollen,[4] existieren zwei Methoden der Kommunikation:

(a) Zwischen den Systemen werden unkontrolliert Nachrichten ausgetauscht, vergleichbar mit dem Versand von Postkarten. Die Systeme müssen selbst für die korrekte Einhaltung der Reihenfolge der Nachrichten sorgen und Kontrollmechanismen vorsehen, ob Nachrichten auch beim anderen Partner angekommen sind.

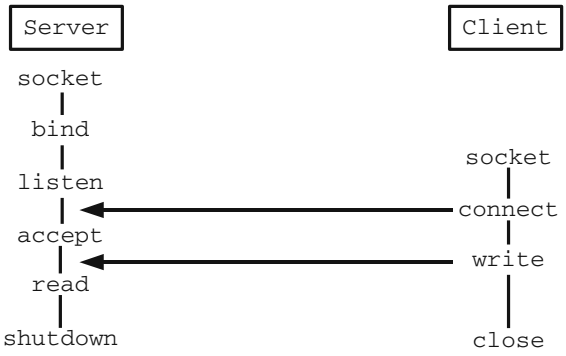
    Diese Kommunikationsart wird beispielsweise durch das Internetprotokoll UDP realisiert.

(b) Zwischen den Systemen werden Verbindungen ähnlich einer Telefonleitung aufgebaut. Versandte Nachrichten kommen garantiert an, so lange die Verbindung existiert. Kontrollmechanismen können in dieser Umgebung sehr viel einfacher realisiert werden.

    Diese Kommunikationsart wird beispielsweise durch das Internetprotokoll TCP realisiert.

[4] Ausführliche Informationen finden Sie beispielsweise in Gilbert Brands, IT-Sicherheitsmanagement, im gleichen Verlag.

Die Einrichtung und Bedienung der Server- und Clientschnittstellen erfolgt mit Hilfe einer Reihe von Betriebssystemfunktionen, die wir kurz erläutern werden. Wir beschränken uns dabei auf die Verwendung der Internetprotokolle (TCP/IP, UDP/IP), und ich setze eine grundsätzliche Kenntnis der Protokolle beim Leser voraus und werde nur rudimentäre zusätzliche Erläuterungen geben können, die Sie bitte ggf. aus anderen Quellen ergänzen.



Funktionsaufrufe im TCP-Modus

18.3.1.1 Anmeldung für eine bestimmte Übertragungsart

Die Anwendung (*sowohl Client als auch Server*) reserviert sich zunächst vom Betriebssystem die gewünschte Übertragungsressource mittels der Funktion

```
typedef SOCKET  int;
SOCKET socket (
  int af       = PF_INET,
  int type,
  int protocol = 0 };
```

Da wir hier nur die TCP/IP-Protokollfamilie behandeln, habe ich gleich die einzusetzenden Konstanten im Funktionskopf angegeben. Über den mittleren Parameter wird der Verbindungstyp vorgewählt.

- Ein Socket für eine UDP-Verbindung wird durch `type=SOCK_DGRAM` spezifiziert,
- ein Socket für eine TCP-Verbindung durch `type=SOCK_STREAM`,

Für die Einrichtung eines UDP-Sockets genügt somit der Funktionsaufruf

```
UDP_Client::UDP_Client(){
    system_init(true);[5]
    sock=socket(PF_INET,SOCK_DGRAM,0);
}//end function
```

Der Rückgabetyp der Funktion ist `int`, das heißt die Anwendung bekommt die eigentliche Struktur eines Sockets nicht zu Gesicht, sondern nur einen „Handle“ darauf. Einerseits macht das die Anwendungen weniger anfällig, da Anwendungsprogramme keine Chance haben, wichtige Daten zu manipulieren, andererseits kann nun jeder, der diese Zahl kennt, über diesen Socket kommunizieren, was insbesondere die Verteilung auf mehrere Prozesse vereinfacht.

Das weitere Schicksal des Sockets ist nun davon abhängig, ob wir eine Client- oder eine Serveranwendung vor uns haben und diese im UDP- oder TCP-Modus arbeiten soll. Beginnen wir mit einer Server-Anwendung, da diese betriebsbereit sein muss, bevor ein Client etwas senden kann.

18.3.1.2 Server-Socket-Programmierung

Der Server-Socket ist nun noch recht „nackt“, das heißt er kennt zumindest noch nicht die Portnummer, unter der die Serveranwendung zu erreichen ist.[6] Da Maschinen mit mehreren Netzwerkeingängen auch mehrere IP-Adressen aufweisen können, ist diese ebenfalls zu spezifizieren. Mit der Funktion

```
int bind(
  SOCKET s,
  const struct sockaddr*  name,
  int namelen
);
```

wird der Socket an eine bestimmte IP-Adresse und einen Port gebunden, wobei die Struktur `sockaddr` für TCP/IP folgendermaßen definiert ist:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
};

struct in_addr { uint32 addr ;};
```

Ist unter dieser Kombination IP-Adresse/Portnummer noch keine Anwendung im Betriebssystem registriert, so verfügt die Serveranwendung nun exklusiv über diesen Anschluss; andernfalls weist `bind(..)` das Ansinnen zurück. Eine typische Serverprogrammierung sieht damit folgendermaßen aus:

---

[5] Manche Systeme wie Windows müssen vor der Anforderung eines Sockets noch entsprechend vorbehandelt werden, was hier durch diesen Methodenaufruf erfolgt. Details entnehmen Sie bitte der Dokumentation Ihres Betriebssystems.

[6] Die Adressierung weist zwei Teile auf: eine Maschinenadresse (IP-Adresse), die eine Netzwerkkarte auf einem System spezifiziert, und eine Anwendungsadresse (Portnummer), mit der man eine bestimmte Anwendung auf dem System ansprechen kann. Portnummern für Server sind meist genormt, um auf beliebigen Maschinen immer einen bestimmten Dienst erreichen zu können.

```
sockaddr_in sa;
sa.sin_family=AF_INET;
sa.sin_port=htons(port);
sa.sin_addr.s_addr=ADDR_ANY;
if(bind(sock,(sockaddr*)&sa,sizeof(sa))==SOCKET_ERROR){
```

Die Programmierung ist für TCP- und UDP-Sockets bis hier hin die gleiche. Ein Client kann später den Server unter der angegebenen Portnummer ansprechen.

Nachdem der Socket nun vollständig initialisiert ist, kann die Serveranwendung ihn aktiv schalten. Dabei ist ein Unterschied zwischen TCP und UDP zu machen, da UDP-Dialoge netztechnisch gesehen nur aus einem Datagramm bestehen, TCP-Dialoge aber eine feste Verbindung erfordern, die gegebenenfalls längere Zeit bestehen bleibt. Für TCP-Verbindungen ist es deshalb sinnvoll, dem Betriebssystem mitzuteilen, wie viele Verbindungen die Anwendung gleichzeitig bearbeiten kann. Dies erfolgt mit der Funktion

```
int listen (SOCKET sock, int anz);
```

Ist die angegebene Anzahl von Verbindungen aktiv, lehnt das Betriebssystem weitere Anfragen ab, ohne erst die Anwendung zu behelligen. Mit der Funktion

```
SOCKET accept (SOCKET sock,
               struct sockaddr*  name,
               int namelen);
```

werden eingehende Verbindungen entgegengenommen und können anschließend bearbeitet werden. Bei einem eingehenden Verbindungswunsch überträgt der Client seine IP-Adresse und seine Portnummer, so dass nun eine eindeutige Kennung von jeweils zwei IP-Adressen und zwei Portnummern für diese Verbindung entsteht. Die Clientangaben werden der Serveranwendung durch den Parameter `name` mitgeteilt. Das Betriebssystem benutzt aber nicht den ursprünglichen Socket, sondern stellt eine Kopie her, auf die die Funktion `accept(..)` einen neuen Handle zurück gibt. Die Serveranwendung kann nun einerseits mit dem neuen Sockethandle eine Kommunikation mit dem Client durchführen, andererseits mit dem alten Handle weiterhin über die `accept`-Funktion feststellen, ob weitere Clients Kontakt aufnehmen möchten.

Die `accept`-Methode ist normalerweise blockierend, das heißt bei einem Aufruf wird die Kontrolle an das rufende Programm erst dann zurückgegeben, wenn tatsächlich eine Clientanmeldung erfolgt. Ein gleichzeitiges Bearbeiten einer Verbindung und das Warten auf eine weitere Clientanmeldung ist somit innerhalb eines Prozesses nicht möglich. Hier kommt nun die in der Einleitung diskutierte Technik zum Einsatz, indem über `fork()` oder `execve`(..) ein neuer Prozess zur Bedienung des Client gestartet wird, während der Primärserver auf weitere Verbindungen wartet.

18.3.1.3 Client-Socket-Programmierung

Für den Client ist die Herstellung einer Verbindung einfacher. Nach Erhalt eines Sockets vom Betriebssystem ruft die Anwendung die Methode

```
int connect(SOCKET sock,
            struct sockaddr FAR*  name,
            int namelen);
```

auf, und zwar einheitlich für UDP- und TCP-Protokolle.

- Im Falle eines UDP-Sockets wird durch den Aufruf der Socket vervollständigt und in allen nachfolgenden Sendebefehlen verwendet, bis ein neuer `connect(..)`-Befehl erfolgt oder der Socket freigegeben wird. Allerdings kann auf diesem Socket nur gesendet werden; für den empfang von Daten muss die Anwendung ihrerseits einen Serversocket öffnen.
- Ein TCP-Socket stellt bei Aufruf der Funktion eine Verbindung zur Zielmaschine her und wird bei Erfolg bis zum Beenden der Verbindung zum Senden und Empfangen von Datagrammen verwendet. Mit Einrichten des logischen Sockets ist der Anwender sicher, dass das Zielsystem empfangsbereit ist.

Das Clientsystem muss lediglich die Portnummer des Serverprozesses und die IP-Adresse des Servers kennen.[7]

18.3.1.4 Der Kommunikationsverlauf

Nach Herstellen der vollständigen Verbindung können bei einer TCP-Verbindung Daten nun mit den Funktionen

```
int read(int handle, char* buf, int size);
int write(int handle, char* buf, int size);
```

ausgetauscht werden. Die Blockgrößen können theoretisch nahe an 64 kByte liegen; größere Datenmengen sind von der Anwendung passend zu segmentieren. Im Standardmodus sind die Schreib- und Lesemethoden blockierend, das heißt die Schreibmethode gibt die Kontrolle erst dann an die rufende Methode zurück, wenn sämtliche Daten in den Sendepuffer übernommen werden konnten, und die Lesemethode meldet sich erst nach Empfang eines kompletten Datensatzes zurück, beziehungsweise wenn die Verbindung beendet wurde.

In vielen Protokollen ist der Kommunikationsablauf vorgeschrieben, und aus dem aktuellen Status der Kommunikation weiß jede Seite, ob ein Lese- oder ein Schreibprozess ansteht. Durch das Protokoll wird ebenfalls festgelegt, wann die Kommunikation beendet ist. Der Socket wird durch die Funktion

```
int shutdown(SOCKET sock, int how);
```

---

[7] Oft sind nur Namen und nicht IP-Adressen von Servern bekannt. Letztere können mit Hilfe der Methode ‘gethostbyname(..)’ ermittelt werden.

für das weitere Senden von dieser Seite geschlossen, während das Empfangen noch weiter möglich ist. Nach Empfang der letzten Nachricht wird der Socket durch

```
int close(SOCKET sock);
```

freigegeben und nimmt im Weiteren auch keine Nachrichten mehr entgegen. Nach Schließen des Sockets, was auch aufgrund eines Leitungsfehlers erfolgen kann (*die Untergrundkommunikation läuft nicht mehr korrekt ab*) blockieren die Methoden nicht mehr und bei einem Lese- oder Schreibversuch wird durch den Rückgabewert 0 oder -1 angezeigt, dass der Socket einseitig geschlossen wurde.

Die Anwendungen müssen intern beim Lesen und Schreiben kontrollieren, ob auch Daten transferiert wurden oder durch die Rückgabewerte ein Schließen des Sockets signalisiert wird. Wie auf ein unerwartetes Schließen eines Sockets reagiert werden muss, ist anwendungsabhängig.

> **Aufgabe.** Die Funktionalitäten haben wir hier mit Verweis auf weiterführende Literatur zwar nur angerissen, so dass Sie sich das Eine oder Andere noch zusammensuchen müssen. Versuchen Sie sich dennoch ein einer Implementation der Klassen `UDP_Client, UDP_Server, TCP_Client` und `TCP_Server`.
>
> Sie benötigen für diese Arbeit nur einen Rechner. Wenn Sie die IP-Adresse `127.0.0.1` verwenden, bleibt das Ganze auf Ihrem Rechner und Sie können zwei gleichzeitig laufende Programme miteinander kommunizieren lassen. Achten Sie aber auf die richtigen Einstellungen Ihrer Firewall ! Wenn keine Verbindung zustande kommt, liegt das möglicherweise nicht an einer falschen Programmierung, sondern einer Blockade durch die Firewall.

18.3.2 Verteilte Objekte

Die Netzwerktechniken ermöglichen es, Code und Daten nun auch maschinenmäßig zu trennen. Die Motivation hierzu kann verschiedene Gründe haben, z.B.:

- Der Anwender soll immer mit dem aktuellen Anwendungsstand arbeiten, ohne laufend Updates laden und installieren zu müssen.
- Die Anwendung ist recht kostspielig und wird selten benötigt, so dass eine fallweise Bezahlung interessanter ist als ein kompletter Erwerb.
- Die Anwendung muss auf umfangreiche Ressourcen zurückgreifen, die man dem Anwender nicht überlassen möchte oder kann.
- ...

Es bieten sich zwei Möglichkeiten an, dies zu realisieren, wobei wir auf bereits diskutierte Techniken zurückgreifen. Ich reiße daher die Vorgehensweise diesmal nur an, ohne selbst in die Details der Programmierung zu gehen. Sie können dies deshalb ebenso wie das folgende abschließende Kapitel als Anregung zu eigenen weiterführenden Arbeiten betrachten.

18.3.2.1 Stellvertretertechnik (CORBA)

Eine Möglichkeit besteht in der Implementation einer Klasse in getrennten Versionen für den Server und den Client. Die Serverversion wird dabei zunächst völlig normal implementiert und getestet. Für den Einsatz in einem Corba-Modell sollte die Klasse von einer Objektfabrikklasse erben, die auch die Technik der „Benutzung neuer Methoden“ (vergleiche entsprechendes Teilkapitel im Kapitel Objektfabriken) unterstützt. Die Serverklasse kann anschließend in einer Objektfabrik zur Verfügung gestellt werden.

Als zweite Komponente stellen wir einen TCP-Server bereit, wobei sich das Klonen des Prozesses als einfachste Version anbietet, so lange verschiedene Serverobjekte nicht miteinander kommunizieren müssen.[8] Pro Serverprozess wird ein Objekt aus der Objektfabrikpalette verwaltet. Bei Aufbau einer Verbindung wird ein Objekt aus der Objektfabrik erzeugt (der Client sendet den Klassennamen), anschließend sendet der Client Funktionsaufrufe mit Daten in Textform und erhält die Antwort des Objektes in der gleichen Form zurück. Der Einsatz der Techniken für Objektfabriken macht den Server unabhängig von den Klassen in der Fabrik, so dass dieses Modell leicht durch neue Klassen erweitert werden kann.

Auf dem Client muss von der Klasse nun lediglich ein Gerüst, also ein Stellvertreter für das Serverobjekt vorliegen. Die Klasse erbt von einer TCP-Clientklasse, die bei Erzeugen des Objektes eine Verbindung zum Server herstellt und dort für die Erzeugung des eigentlichen Objektes sorgt, bei der Vernichtung auch den Serverprozess samt Objektinstanz beendet.

Die Gerüstklasse stellt sämtliche notwendigen Attribute sowie sämtliche öffentlichen Methoden zur Verfügung. Ein Methodenaufruf wird samt der notwendigen Parameter in einen String übersetzt und an den Server übergeben, dort ausgewertet und das Ergebnis wieder zurück übersetzt. Für die Clientanwendung ist dieser Vorgang vollständig transparent, d.h. abgesehen von einem möglichen Zeitverlust für die Übertragung im Netzwerk besteht kein Unterschied zu einer direkten Ausführung auf dem Clientsystem.

Dieses Modell können Sie schrittweise verfeinern, indem Sie beispielsweise die Kommunikationskanäle nicht objektweise, sondern prozessweise öffenen und alle Objekt über die gleiche Verbindung mit dem Server kommunizieren lassen. Es bietet sich hier eine Erweiterung der Objektfabriktechnik und eine Erzeugung sämtlicher Klassen auf dem Client mit Hilfe einer Objektfabrik an. Die Objektfabrik baut bei der Instanziierung die Kommunikation zum Server auf und versorgt neu erzeugte Objekte mit den Socketinformationen. Bei vielen Objekten mit kurzer Lebensdauer werden so Ressourcen auf dem Server eingespart (dieser verwaltet nun alle Objekte in einem Prozess und benötigt weniger Speicherplatz; für die Verwaltung mehrerer Objekte und die Differenzierung zwischen den Objekten müssen Sie ihn aber erst

[8] Sollten Sie beispielsweise auf die Idee kommen, einen Gruppeneditor zu implementieren, bei dem jedes Gruppenmitglied online alle Änderungen sieht, wechseln Sie einfach zu einer Thread-Technik mit einem gemeinsamen Speicherbereich.

mal softwaremäßig fit machen) sowie weniger Zeit für den Auf- und Abbau von Verbindungen benötigt.[9]

18.3.2.2 Nachladen von Code

Werden von den Objekten größere Datenmengen verwaltet, wird die Stellvertretertechnik aufgrund der relativ langsamen Übertragungswege ungünstig. Statt dessen kann es sich anbieten, Kodeteile nachzuladen.

Auch das ist relativ einfach mit Hilfe der Objektfabriktechnik zu erledigen. Hierzu werden die Anwendungen zunächst völlig normal entwickelt, wobei das Gros des Kodes allerdings nicht direkt in die Anwendung eingelinkt, sondern in Form von DLLs bereitgestellt wird, die allerdings nicht mit der Anwendung ausgeliefert werden. Die Objektfabrik wird nun dahingehend erweitert, dass nicht nur direkt instanziierbare Klassen registriert werden, sondern auch auf Servern verfügbare Klassen nebst den Namen der zugehörenden DLLs. Das kann fabrikintern durch eine Datenbank realisiert werden, die abgefragt wird, sobald ein Klassennamen nicht in der normalen Registratur gefunden werden kann.

Wird nun ein Objekt einer solchen Klasse benötigt, wird die DLL durch die Objektfabrik vom Server abgerufen, als temporäre Datei gespeichert und mit den vorgestellten Techniken in die Anwendung eingebunden. Anschließend können die Objekte wie diskutiert instanziiert werden. Bei Beenden der Anwendung werden temporäre Dateien gelöscht, so dass die DLL wieder vom Clientsystem verschwindet. Der Ladeprozess kann auf dem Server überwacht und dem Anwender in Rechnung gestellt werden.

18.4 Parallele und massiv parallele Prozesse

Die Aufteilung einer Anwendung in Prozesse oder Threads führt zwar formal zu einer Parallelisierung von Aufgaben, in den meisten Fällen wird die Technik jedoch nur genutzt, um Aufgaben zu entflechten und den Code übersichtlich zu halten (vergleiche auch die Anmerkungen zum Einsatz von Exceptions). Bei vielen rechenintensiven Aufgaben bietet sich aber eine tatsächliche Parallelisierung an, sofern Teilberechnungen hinreichend scharf voneinander abgrenzbar sind.

Die Aufteilung einer Anwendung auf verschiedene Prozesse oder Threads unterliegt der Analyse des Problems und der Anwendungsgestaltung durch den Anwendungsprogrammierer. In Umgebungen mit mehreren Prozessoren oder mit vielen durch ein Netzwerk verbundenen Maschinen können die Einzelaufgaben auf unterschiedliche Prozessoren verteilt und tatsächlich parallel ausgeführt werden. Die entwickelten Programmiertechniken reichen hierzu vollkommen aus, da die Verteilung Angelegenheit des Betriebssystems ist.

---

[9] Wenn Sie Threads einsetzen, gehören die Kommunikationsfolgen allerdings in ein Monitorobjekt, siehe oben in diesem Hauptkapitel.

Trotzdem muss sich der Anwendungsprogrammierer natürlich mit der Betriebsumgebung vertraut machen. Werden Speicherbereiche beispielsweise von verschiedenen Threads genutzt, die aber auf verschiedenen Maschinen laufen, so kann es durch Synchronisierungsvorgänge der Speicherinhalte zu Laufzeiten kommen, die schlechter sind als auf einer Einzelprozessorumgebung. Das Betriebssystem wird als Gegenmaßnahme Gruppierungsfunktionen für die einzelnen Threads anbieten, die vom Anwendungsprogrammierer sinnvoll eingesetzt werden müssen.

Zukünftige Rechnergenerationen bieten möglicherweise noch weitere Möglichkeiten der Parallelisierung, die auf Mikroebene einzelne Anweisungsblöcke betreffen. Der Leser stelle sich beispielsweise eine CPU vor, die aus mehreren tausend sehr einfachen, aber dafür auch sehr schnellen Einzel-CPUs besteht (eine so genannte GRID-CPU). Die in vielen Anwendungen auftretende Berechnung von Skalarprodukten

```
for(i=0 ; i<max ; i++)
  sum=sum + a[i]*b[i];
```

könnte in solchen Umgebungen kaskadiert werden, d.h. die CPU erledigen die Arbeit durch

```
        -------->  Zeit
C0   a[0]*b[0]   +{C1}  +{C2}  ...
C1   a[1]*b[1]   ...
C2   a[2]*b[2]   +{C3}  ...
C3   a[3]*b[3]   ...
...
```

Während die Laufzeit in allen heutigen Systemen linear mit der Schleifengröße steigt, könnten solche Systeme die Aufgabe in erledigen. Voraussetzung hierfür wären allerdings Bussysteme, die in der Lage sind, die Daten auch in der notwendigen Geschwindigkeit in die GRID-CPU einzuspeisen (optische Busse vermögen dies voraussichtlich).

Solche Techniken sind natürlich nur einsetzbar, wenn die Rechnungen innerhalb der Anweisungsblöcke auch voneinander unabhängig durchgeführt werden können. Sprachtechnische Erweiterungen wären sicher hierzu notwendig, beispielsweise

```
for(i=0 ; i<max ; i++) cascade (sum=0) {
  sum=sum + a[i]*b[i]; }
```

um die Parallelisierung für die Variable `sum` zu ermöglichen, oder

```
for(i=0 ; i<max ; i++) cascade (sum=0) serial (sum) {
  sum=sum*c[i] + a[i]*b[i]; }
```

um neben der Parallelisierung noch dafür zu sorgen, dass die Zwischenergebnisse nicht in beliebiger Reihenfolge weiterbearbeitet werden können.

Ob eine derartige Hardware realisiert wird, bleibt abzuwarten. Trotzdem ist es sicher jetzt schon reizvoll, Modelle für Systemsoftware zu entwickeln. Wie in heutigen parallelen Prozessen kann man davon ausgehen, dass auch Grid-CPUs

asynchron betrieben werden, d.h. jede CPU ihren Job unabhängig von den anderen erledigt. Das muss natürlich nicht heißen, dass einzelne CPUs Sonderaufgaben in der Koordinierung übernehmen. Sinnvoll ist dann möglicherweise auch, die Datenbusse seriell statt parallel aufzubauen, um Datenworte auch sicher zusammen zuhalten (ein heutiger 64-Bit-Parallelbus kann beispielsweise durch 64 parallel arbeitenden serielle Einzelbusse ersetzt werden, von denen jeder als optischer Bus in der Lage sein sollte, die Daten mindestens in der Geschwindigkeit heutiger CPU-Verarbeitungstakte zu transportieren). Welche Informationen müssen einem Datenwort (das als „Superdatenwort“ dann auch mit wesentlich mehr Bits als 64 aufgebaut werden kann) zugeordnet werden, damit eine Kaskadierung funktioniert? Mit der Anregung zu solchen Überlegungen und dazugehörenden Versuchen, die sicher vieles von der im Laufe des Buches diskutierten Technik benötigen und zeitlich durchaus zu einer Langzeitbeschäftigung ausarten können, und der Hoffnung, dass Ihnen der Inhalt Spaß gemacht hat und Lehrreiches und Brauchbares enthalten hat, möchte ich dieses Buch schließen