Themen zur kurzen Wiederholung, Näheres siehe Buch

Prozesse, Threads

Prozesse

Multitask- oder Multiprozess-Systeme haben privaten Speicher
für Datensegment, Heap und Stack, durch den Einsatz von mächtigeren
MMUs auf größeren Prozessoren ist es deshalb möglich, fremde Datenzugriffe
zu unterbinden.

Für Prozesse geht man davon aus, dass die MMU des Systems ihnen virtuell
unbeschränkt Stack und Heap zuteilen können.

Ein Prozess/Thread befindet sich normalerweise in einem der folgenden Zustände:
• READY - Der Prozess kann die CPU nutzen.
• BLOCKED - Der Prozess befindet sich in einem blockierenden Zustand
und wartet auf Freigabe.
• HELD - Der Prozess hat ein SIGSTOP-Signal erhalten und wartet auf ein
SIGCONT-Signal oder auf Prozesstermination über ein Signal.
• WAIT - Der Prozess wartet auf Statusinformationen von einem oder mehreren
seiner Sohnprozesse.
• DEAD - Ein Zombie-Prozess.

Virtueller Adressraum
Ein Adressraum ist die abstrakte Repräsentation eines Speichers. Ein solcher
virtueller Adressraum ist unendlich groß und die einzelnen Speicherzellen
sind durchgehend nummeriert (linearer Adressraum).

Threads
Ein Thread, auch Aktivitätsträger oder Laufzeitfaden genannt, ist die abstrakte
Repräsentation eines Prozessors. Je nach Laufzeitmodell ist es möglich,
mehrere dieser virtuellen Prozessoren innerhalb des Adressraums eines Prozesses
zu erzeugen und gleichzeitig ablaufen zu lassen. Der erste Thread, der
in einem neu erzeugten Prozess läuft, wird Main-Thread genannt.

Zwar besitzen Threads normalerweise jeweils ein eigenes Stack-Segment,
jedoch sind Daten-, Heap und Code-Segmente gemeinsam.
Daher können z.B. ohne weiteres Daten anderer Threads direkt aufgerufen
werden, auch Callbacks sind möglich, da der Speicher nicht abgeschottet ist.

Prozesskommunikation

• über Dateien im File-System,
• über die noch vorzustellenden IPCs,
• über einen geteilten Speicher (Shared-Memory).

Threadkommunikation

Die Kommunikation zwischen den einzelnen Threads kann über gemeinsamen Speicher abgewickelt werden, alle Adressen sind allen Threads bekannt. So können z.B. einfache globale Variable verwendet werden. Auch gegenseitige Aufrufe von Programmteilen (call, callback) sind üblich. Man spricht von einer speichergekoppelten Kommunikation.

Threads erzeugen

```
#include <pthread.h>
int pthread_create(pthread_t *neueThreadIDPtr,      //Thread_ID Rückgabe
   const pthread_attr_t *attrPtr,                   //Attribute des neuen Threads
   void * (*startFuntion) (void *),                 /ähnlich main im Prozess
   void  * argPtr)                                  //Zeiger auf Argumente
                                                    //Rückgabe: EOK (0) OK, sonst Fehler
                                                    //Bemerkung: Wenn attrPtr NULL, werden
                                                    //Defaultattr. für erz. Thread genommen
```

Dabei ist
.... void * (*startFunction) (void*) ...
der Funktionszeiger der Start-Funktion im neuen Thread, quasi die main im neuen Thread.

Prozesse forken

Durch die Funktion fork wird ein neuer Prozess erzeugt, indem der laufende, so genannte Vaterprozess vollständig kopiert und dann in zwei laufende Prozesse verzweigt wird. Da Vater- und Sohn-Prozess (die Kopie) anhand des Funktionsergebnisses unterschieden werden können, kann bei entsprechender Programmanweisung in eigene Programme verzweigt werden.
Der neue Prozess teilt den gleichen Code mit dem aufrufenden Prozess und erbt eine Kopie aller Prozessdaten des rufenden Prozesses.

Syntax:

```
pid_t fork()        //-1 falls Fehler,
                    //0 im Sohnprozess,
                    //sonst ProzessID des Sohnes
```

Beispiel:

```
int main(void)
{
  int context;                  //Dummy Kontext, kommt später
  ...                           //Dinge initialisieren
  if (fork() == 0)
  {
   sOwnCout<<”adresse von context-Kind" << &context  << endl;
   ...;                         //ab hier läuft erster KindProzess
  }
  else if (fork() == 0)
  {
   sOwnCout<< “adresse von context-Kind" << &context << endl;
   ...;                        //ab hier läuft zweiter KindProzess
  }
  else
  {
   sOwnCout<<"adresse von context-Vater" << &context << endl;
   ...;                        //ab hier läuft Vaterprozess weiter
  }
}
```

IRunnable

Basis jedes Prozesses, jedes Threads und jeder Komponente soll das abstrakte Interface IRunnable sein. Damit gibt es ein gemeinsames Verständnis und Interface für alle Prozesse und Threads.
Jeder neue Thread (auch im Sinne von main-Thread) muss von IRunnable erben beziehungsweise dieses Interface über ein Referenzobjekt nutzen, die Interface-Definitionen mit Implementierungen füllen und die Schnittstellen bedienen.

```
class IRunnable                 // INTERFACE DEFINITION
{
 public:
    //Init and startup actions in thread context. This method will be
    // called precisely once, within the thread context, prior to the
    // call of the 'run' method of this IRunnable object. This allows
    // the IRunnable to perform any init operations that may be
    // required within the execution context of the thread.
   virtual void init(void) = 0;

   //  Perform the work of the IRunnable object of this thread.
   // This method is not expected to return until either the work of
   // the thread is completed, or until the stop() method of this
   //  object has been called.
   virtual void run(void) = 0;

   // Trigger a shutdown of this thread. The run() method is expected
   // to react to this call by exiting. This method may not be called
   // from within the thread context, for that reason cleanup
   // operations are done separately (see the cleanup method).
   virtual void stop(void) = 0;

   // Perform cleanup actions within the thread context. This method
   // is called immediately following the exit from the run() method,
   // within the thread context. This is where cleanup actions for
   // this thread should be performed.
   virtual void cleanup(void) = 0;

   // request to the component to prepare the necessary steps due to
   // possible shutdown    */
   virtual void prepareForShutdown(void) = 0;
   // request for resume
   virtual void reRun() = 0;
   // request: the component should go to the standby mode
```

```
    virtual void goToStandby() = 0;
};
```

Ein Thread wird als Träger eines IRunnable-Objekts angesehen. Um

einen Thread zu generieren, wird deshalb eine Thread-Klasse definiert, die ein IRunnable-Objekt als Parameter übergeben bekommt (und dieses trägt (carrier)). Die Klasse ruft dann die üblichen IRunnable-Methoden auf, beispielsweise init und run.

Beispiel CThread:

```
class CThread          //Utility class to encapsulate thread  control.
{
public:
  enum EPriority       //only 16 priorities are supported
    {
      PRIORITY_IDLE,
      PRIORITY_ABOVE_IDLE,
      PRIORITY_BELOW_LOW,
      PRIORITY_LOW,
      PRIORITY_ABOVE_LOW,
      PRIORITY_BELOW_NORM,
      PRIORITY_NORM,
      PRIORITY_ABOVE_NORM,
      PRIORITY_BELOW_HIGH,
      PRIORITY_HIGH,
      PRIORITY_ABOVE_HIGH,
      PRIORITY_BELOW_CRITICAL,
      PRIORITY_CRITICAL,
      PRIORITY_ABOVE_CRITICAL,
      PRIORITY_BELOW_REALTIME,
      PRIORITY_REALTIME
    };
  CThread(IRunnable & runnableObj,          //runnableObj that runs in thread
     const char * name,                     //name of the thread
     Int32 stackSize,                       //size of the stack
     EPriority prio,                        //thread priority
     bool asMainThread = false );           //if runnable runs in main thread
  ~CThread();                               //Destructor
  void start(void);                         //start the object as a thread
  void join()                               //join the thread
  {
    pthread_join(mOSThreadID, NULLPTR);
  }

  TID getThreadID(void)                     //return the thread ID
  {
    return mOSThreadID;
  }

  static void sleep(Int32 mSecs);           //Sleep without consuming CPU
  static TID getThreadID(const char * name);  //ret ID,(0 if not exist)

  static TID getCurrentThreadID(void)  //my own
  {
    return pthread_self();
  }

  static EPriority getCurrentThreadPrio(void);
  static const char * getThreadName(TID tid);
  static const char * getCurrentThreadName(void);

  //get the thread name, NULLPTR if thread does not exist
  static const char * getThreadName(TID tid);
```

```
  static const char * getCurrentThreadName(void);  //my own

  //get the pointer to the runnable object
  static IRunnable * getRunnablePtr(TID tid);
  static IRunnable * getCurrentRunnablePtr(void);

 private:
  friend void init(void);                    //C-method to init tsd
  friend void *threadProc(void *);           //thread-startmethod
  void run();                                //activate thread in its context
  IRunnable* mRunnablePtr;                   //object with the thread activity
  const char*  mName;                        //Name
  Int32 mStackSize;                          //stacksize
  EPriority mPrioBase;                       //priority of the thread
  //remember if new thread be started later
  bool mAsMainThread;
  pthread_t mOSThreadID;                     //ID of thread
  static VoidPtrVector sThreads;             //list of all threads
  static CMutex sMutex;                      //protect again concur.accesses
  static pthread_key_t sThreadKey;           //to store the thread-local data
  static pthread_once_t sInit;               //ensure init called only once
  static Int32 sIndex;                       //for the traversing the list
};
```

Beispielaufruf als Thread:

```
CAudiodiscplayerComponent audiodiscplayer
  (CContext::getAudiodiscplayerContext(),1,1);
                                  //Erzeugen Runnable Obj.
CThread audiodiscplayerThread
  (                               //Erzeug.d.Träger-Threads
    audiodiscplayer,                    //Objekt als Parameter
    CContext::getAudiodiscplayerContext().getContextNamePtr(), //Name
    CContext::AUDIODISCPLAYER_STACK_SIZE,     //stack size
    CContext::AUDIODISCPLAYER_PRIORITY
  );                              //Prio
  audiodiscplayerThread.start();
```

Als Prozess:

```
if (0 == fork())
{
  CTunerComponent tuner( ...);
  CThread c(tuner, "Tuner", 1024, CThread:: PRIORITY_NORM, true);
  c.start();
}
```

Mit dieser Klasse haben wir die Möglichkeit, Threads und Main-Threads
einheitlich zu behandeln.

```
//declaration of static variables of the class CThread
VoidPtrVector CThread::sThreads(5,5);
CMutex                    CThread::sMutex;
pthread_key_t             CThread::sThreadKey;
pthread_once_t CThread::sInit = PTHREAD_ONCE_INIT;
Int32         CThread::sIndex = 0;

void init(void)               //method that will called only once
{
  pthread_key_create(&CThread::sThreadKey, NULLPTR);
}

void* threadProc(void* threadPtr)     //Thread start method
{
  pthread_once(&CThread::sInit, init);
  if (NULLPTR != threadPtr)
  {
    CThread *thisThreadPtr = reinterpret_cast<CThread*>(threadPtr);
    pthread_setspecific(                          //set thread-local for fast
      CThread::sThreadKey,                        //calling the get-methods
      reinterpret_cast<void *> (thisThreadPtr));
      thisThreadPtr->run();                       //call working method of thread
  }
  return NULLPTR;
}
CThread::CThread(IRunnable & runnableObj,         /object that run in a thread
  const char * name,                              //name of the thread
  Int32 stackSize,                                //size of the stack
  EPriority  prio,                                //thread priority
  bool asMainThread)                              //if runnable run in main thread
 : mRunnablePtr(&runnableObj),
  mName(name),
  mStackSize(stackSize),
  mPrioBase(prio),
  mAsMainThread(asMainThread)
{
 ASSERTION(asMainThread == false && stackSize > 0);
 ASSERTION(mName != NULLPTR);
 sMutex.take();                                   //add this thread into the list
 sThreads.add(this);
 sMutex.give();
}

CThread::~CThread()                               //destructor
{
  sMutex.take();
  sThreads.removeElement(this);
  sMutex.give();
}

void CThread::start(void)                         //start thread
{
  Int32 realPrio;
  int ret;                                        //int bcs. return of pthread-calls
  struct sched_param threadparam;
  int max = sched_get_priority_max(SCHED_RR);     //calc right prio
  int min = sched_get_priority_min(SCHED_RR);
  realPrio = min + (max - min)/(PRIORITY_REALTIME + 1)*mPrioBase;
```

```
  if (false == mAsMainThread)
  {
    pthread_attr_t attribute;                                   //Setting thread attributes
    pthread_attr_init(&attribute);                              //Set structure to default
    pthread_attr_setinheritsched(&attribute, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attribute, SCHED_RR);
    pthread_attr_getschedparam(&attribute, &threadparam);       //+set prio
    threadparam.sched_priority = realPrio;
    pthread_attr_setschedparam(&attribute, &threadparam);
    ret = pthread_create(&mOSThreadID, &attribute, threadProc, this);
    ASSERTION (ret == 0 );
    pthread_attr_destroy(&attribute);                           //destroy the attribut
  }
  else
  {                                                 //thread object started as main thread - i.e. in this context
    mOSThreadID = getCurrentThreadID();                         //threadID of mainThread
    int policy;                                                 //change sheduling and prio
    pthread_getschedparam(mOSThreadID, &policy, &threadparam);
    threadparam.sched_priority = realPrio;
    ret = pthread_setschedparam(mOSThreadID, SCHED_RR, &threadparam);
    ASSERTION(ret == 0);
    run();                                                      //call run-method directly
    //we do not leave method because it is the loop in main thread
  }
  (void) ret;                                                   //avoid ünused variable",
}                                                               //case ASSERTION not enabled

void CThread::run(void)                                         //main method of thread
{
  mRunnablePtr->init();
  mRunnablePtr->run();
  mRunnablePtr->cleanup();
}

void CThread::sleep(Int32 mSecs                                  //Sleep w/o consuming CPU
{
#ifdef WIN32
  Sleep(mSecs);                                                  //measure in nano seconds
#endif
#ifdef __GNUC__
  usleep(mSecs*1000);                                            //measure in nano seconds
#endif
}

TID CThread::getThreadID(const char * name)                     //get the thread ID
{
  TID tid = 0;
  sMutex.take();
  Int32 i;
  for (i = 0; i < sThreads.getSize(); i++)                      //run list +compare name
  {
    CThread* threadPtr = reinterpret_cast<CThread *> (sThreads[i]);
    if (0 == strcmp(name, threadPtr->mName))
    {
      tid = threadPtr->mOSThreadID;
      break;
    }
  }
  sMutex.give();
  return tid;
}
```

```
const char*  CThread::getThreadName(TID tid)                    //get the thread name
{
  const char *ptr = NULLPTR;
  sMutex.take();
  Int32 i;                                                      //run list+compare ID
  for (i = 0; i < sThreads.getSize(); i++)
  {
    CThread* threadPtr = reinterpret_cast<CThread *> (sThreads[i]);
    if (tid == threadPtr->mOSThreadID)
    {
      ptr = threadPtr->mName;
      break;
    }
  }
  sMutex.give();
  return ptr;
}

CThread::EPriority CThread::getCurrentThreadPrio(void)
{
  int policy;
  sched_param param;
  int max = sched_get_priority_max(SCHED_RR);                   //calc right prio
  int min = sched_get_priority_min(SCHED_RR);
  pthread_getschedparam(pthread_self(), &policy, &param);
  //calculate the priority
  int prio=(param.sched_priority-min)*(PRIORITY_REALTIME+1)/(max-min);
  if (prio <  PRIORITY_IDLE)
  {
    return  PRIORITY_IDLE;
  }
  else if (prio > PRIORITY_REALTIME)
  {
    return PRIORITY_REALTIME;
  }
  else
  {
    return static_cast<EPriority>(prio);
  }
}

const char *CThread::getCurrentThreadName(void)
{
  CThread* currentPtr =
    reinterpret_cast<CThread*> (pthread_getspecific(sThreadKey));
  return currentPtr->mName;
}

IRunnable * CThread::getRunnablePtr(TID tid) //get ptr to runnable obj
{
  IRunnable *ptr = NULLPTR;
  sMutex.take();
  Int32 i;
  for (i = 0; i < sThreads.getSize(); i++)
  {
    CThread* threadPtr = reinterpret_cast<CThread *> (sThreads[i]);
    if (tid == threadPtr->mOSThreadID)
    {
      ptr = threadPtr->mRunnablePtr;
      break;
    }
  }
```

```
  sMutex.give();
  return ptr;
}

IRunnable * CThread::getCurrentRunnablePtr(void)
{
  CThread* currentPtr
    = reinterpret_cast<CThread*> (pthread_getspecific(sThreadKey));
  return currentPtr->mRunnablePtr;
}
```

Bei genauer Betrachtung fällt auf, dass die Liste aller Threads in einer Variablen des eigenen Typs {VoidPtrVector} aufbewahrt wird. Sie wird mit einer Startgröße und einem Inkrement vereinbart, so dass die Liste, falls nötig, wachsen könnte.

Dies widerspricht zwar den selbst aufgestellten Regeln, keine Vektoren zu verwenden, da die Vektorliste jedoch mit zum Start-Up des Systems erzeugt wird, ist das mögliche dynamische Verhalten unkritisch und messbar.

Mutex, Semaphor

Manchmal ist es notwendig, dass mehrere Prozesse oder Threads auf gemeinsame Ressourcen zugreifen. Damit die Zugriffe von verschiedenen Teilnehmern nicht durcheinander gehen, müssen sie synchronisiert werden. Es wäre zum Beispiel unlesbar, wenn ein Drucker gleichzeitig und durcheinander von zwei parallelen Prozessen verwendet würde.

Eine kritische Ressource kann auch ein Datenbereich, Queues, Shared-Memory, ein SW-Modul, ein Treiber oder ein paar Codezeilen mit globalen Datenzugriffen sein, die nicht parallel von mehreren Prozessen aufgerufen werden dürfen.

Der Teil des Codes, der wegen der Datenzugriffe gegen verschachtelten Zugriff geschützt werden muss, wird kritischer Abschnitt genannt.

Kritische Abschnitte

Schwieriger sind kritische Abschnitte. Dies sind Codeteile, die gemeinsame Daten verändern können. Alle Funktionen z.B., die von mehreren Threads genutzt werden können, müssen ”threadsafe“ sein.
Der Begriff **threadsafe** bedeutet, dass alle konkurrierenden Zugriffe auf Daten des gemeinsamen Heaps oder des gemeinsamen Datensegments serialisiert werden müssen, sie müssen hintereinander erfolgen. Erst muss der Thread, der die Codestelle mit veränderbaren Daten betreten hat, seine Zugriffe abgeschlossen und die Stelle verlassen haben, damit dann der Nächste diesen Bereich betreten darf.
Man spricht von Synchronisierung der Zugriffe.

Reentrant heißt, dass man erlaubt, dass eine Funktion von einem Thread bereits betreten wurde und benutzt wird und zugleich ein anderer Thread diese Funktion ebenfalls betreten darf. Durch Scheduling und Prioritäten kann sogar der spätere Thread den früheren in der Funktion überholen. Dies bedeutet, dass alle Daten dieser Funktion entweder auf dem jeweils privaten Stack liegen oder konstant sein müssen, die Funktion hat keinen kritischen Bereich. Lägen die Daten auf dem Heap oder wären global im Datensegment, so zerstörten sich die Threads ggf. gegenseitig die Daten.
Die Schutzmechanismen sind eigentlich die Gleichen wie die für den direkten Schutz von gemeinsamen Daten, jedoch ist es ein alltäglicher und sehr schwer zu findender Fehler, wenn Programmierer eine Funktion eines Threads nicht threadsafe oder reentrant implementiert haben.
Oft wissen sie zum Zeitpunkt der Implementierung auch noch nicht, dass ihr Modul einmal eine solche Verwendung finden wird und mehrere Threads darauf zugreifen werden.
Tatsächlich können Treiber, Hilfsfunktionen, Ausgabe- und Debug-Funktionen von verschiedenen Threads parallel aufgerufen werden und jedes globale Datum kann verändert werden.

Man kann sich einen Semaphor wie eine rote Ampel vorstellen, die eine gemeinsame Kreuzung, die aus mehreren Richtungen befahrbar wäre, schützt. Der Erste, der an die Kreuzung heranfährt, bekommt den Semaphor und darf ihn auf {lock} oder {take} (= rot) stellen. Dann fährt er durch die Kreuzung. Erst wenn er nach Verlassen der Kreuzung wieder den Semaphor freigibt (unlock} oder {give}, bildlich auf grün stellt, können andere Fahrzeuge versuchen, den Semaphor für sich zu bekommen.

Einen Semaphor darf eigentlich jeder wieder freigeben, also auch andere Prozesse oder Threads, das Ampelbeispiel hinkt hier etwas. Besser ist das Bild eines amerikanischen Briefkastens, bei dem der Briefträger nach Hineinlegen der Briefe den roten Hebel nach oben stellt.

Der Kunde, der die ganze Zeit auf dieses Signal gewartet hat, holt sich daraufhin seine Briefe aus dem Kasten und stellt den Hebel wieder herunter, gibt den Briefkasten also wieder frei.

Ein Semaphor hat deshalb auch Signalwirkung. Typische Anwendung ist zum Beispiel die Rollenverteilung Produzent (z.B. Interrupthandler) und Konsument (Treiber). Der Konsument wartet auf einen Semaphor, der vom Produzenten bekannt ist. Wenn etwas verfügbar ist, wird der Semaphor freigegeben.

Semaphore und Mutexe sind demnach Variablen, die, da sie von verschiedenen Tasks abgefragt, blockiert oder freigegeben werden, nur vom Betriebssystem verwaltet werden können.

Auch führt das Betriebssystem eine Warteliste von blockierten Prozessen oder Threads, die nach der Freigabe der Semaphore vom Scheduler prioritätengerecht bedient werden.

Zur Synchronisierung müssen die Semaphore allen Beteiligten bekannt sein.
Semaphor, symbolisch:

```
int s=1;                        // auf frei initialisieren
...
P(s):
  P: if (s == 0) goto P         // warten bis frei, dann take das. Hier
                                // ist eine Polling Schleife!!
     else s = s-1;              // blockieren
...                             // die gewünschte Ressource nutzen
V(s):     s = s+1               // freigeben  give
```

Posix:

```
int sem_init(sem_t * semPtr, int shared, unsigned int value);
                // semPtr Zeiger auf ein Semaphor
                // shared != 0 Semaphor kann von
                // mehreren Prozessen verwendet werden.
                // value Startzähler des Semaphors
int sem_wait(sem_t * semPtr);
```

```
                // Der Zähler wird versucht zu erniedrigen. Falls
                //  Wert <= 0, wird rufender Thread blockiert.
int sem_trywait(sem_t * semPtr);
                // Versuch, Semaphor zu holen. Es wird
                // 0 zurückgegeben, falls alles OK ist
int sem_timedwait(sem_t * semPtr, const struct timespec *abs_timeout)
                // Warten mit Timeout. Nur POSIX-draft,
                // wird aber von QNX unterstützt
int sem_post(sem_t * semPtr);
                // Inkrement. des Semaphors, Freigabe
int sem_destroy(sem_t * semPtr);
                // Sem.löschen
```

OO-Wrapper

```
#include <pthread.h>

class CBinarySemaphore
{
 public:
   CBinarySemaphore(bool isLocked = true, bool isProcessShared = true);
   ~CBinarySemaphore();
   //waitForEver=true task waits until sem is free
   bool take(bool waitForEver = true);
   //timeOut is time in millisecs until task returns
   //if -1 the task will wait until sem is free forever
   //return   true if the lock is successfully done
   bool takeWithTimeOut(int timeOut);
   //unlock the semaphor
   void give(void);
 private:
   pthread_mutex_t mMutex;     //mutex
   pthread_cond_t mCondition;  //condvar to implement take with timeout
   int mCounter;               //counter(0/1)-Int32 used for efficiency
};
```

Mutex:
Mutexe sind exklusive Semaphore, d.h., nur der Prozess oder Thread darf den Semaphor freigeben, der ihn auch blockiert hat. Hier stimmt das Ampelbeispiel sehr gut.
Ein Mutex (mutual exclusion = "`wechselseitiger Ausschluss"') wird meistens verwendet, um einen kritischen Abschnitt zu schützen. Der Abschnitt wird von dem Prozess geschützt, der ihn gerade betritt bzw. bereits betreten hat. Dementsprechend ist es sinnvoll, dass ein Mutex auch nur durch den Prozess/Thread freigegeben werden kann, der den Mutex vorher genommen hat.

m Gegensatz zum Semaphor "`gehört"' der Mutex dem Task, der ihn kreiert hat. Dies hat zum einen zur Folge, dass Mutex-Locks schneller als Semaphoren-Locks sind, zum anderen können sie nur zwischen Prozessen synchronisieren, wenn sie die Lokalisierung im Shared-Memory erlauben. Außerhalb des Shared-Memory kann ein fremder Prozess die Mutex-Variable nicht adressieren.

Mutex-Variablen können rekursiv verwendet werden, das heißt, ein Thread kann eine Mutex-Variable mehrfach sperren. Sie muss dann genauso oft wieder freigegeben werden. Dies ist sinnvoll, wenn innerhalb eines Threads in einer Methode durch die Aufrufe weiterer Funktionen in unterschiedlicher Reihenfolge die geschützte Ressource mehrfach belegt werden kann.

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_attr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t * mutex);
int pthread_mutex_lock(   pthread_mutex_t *mutex);          //blockiert falls
                                                            // mutex gelockt
int pthread_mutex_unlock( pthread_mutex_t *mutex);          //freigeben
int pthread_mutex_trylock(pthread_mutex_t *mutex);          //nicht blockieren
```

OO-Wrapper

```
class CMutex
{
public:
  // Create mutex semaphore
  //isFull: initial state of Mutex. false means empty
  //isProcessShared: true, mutex can be accessed by several processes
  CMutex(bool isLocked = false, bool isProcessShared = false);
  ~CMutex();

  //If mutex is locked by other task, our task will be queued
  //waitForEver: if true task will wait until mutex is free
  //return true if lock was successfully done
  bool take(bool waitForEver = true);                       //lock the mutex

  void give(void);                                          //free the mutex

private:
  CMutex(const CMutex& rhs);                                //to avoid misuse
  CMutex& operator=(const CMutex& rhs);
  pthread_mutex_t mMutex;                                   // mutex
}; // end of class CMutex

inline void CMutex::give(void)                              //unlock the mutex
{
  pthread_mutex_unlock(&mMutex);
```

}

Socket, siehe Buch

Shared Memory

Auch bei getrennten Prozessen mit privaten Speicherbereichen kann
gemeinsamer Speicher, Shared-Memory, vereinbart werden.
Zu beachten ist dabei,
dass in jedem Prozess einzeln dieser Speicher vereinbart werden muss,
dass in jedem Prozess der Speicher unter anderen Adressen liegen kann,
man also ggf. Zeiger umrechnen muss (außer beim forken),
dass beim Zugriff auf den Speicher auf Zugriffskollisionen mit anderen Prozessen, die diesen Speicher teilen, geachtet werden muss, siehe unten,
dass der Bereich ggf. gegen swappen (Auslagern des Speichers auf Festplatte) gelockt werden muss und dass dieser Speicher nicht Teil eines der vorgestellten Segmente ist, demnach selbst verwaltet werden muss.

1. Erzeugen bzw. Holen des Shared-Memory-Objekts:

```
int shm_open(const char* name, int oflag, mode_t mode)
  //name: Name des benannten Shared-Memory.
  //oflag: entweder O_RDONLY oder O_RDWR
  //zusammen mit O_CREAT, O_EXCL oder O_TRUNC.
  //Mit O_CREAT | O_EXCL (exkl. Erzeugung) wird das Objekt erzeugt,
  //wenn es noch nicht existiert. Ein Fehler EEXISTS wird zurück-
  //gegeben, wenn das Objekt schon angelegt wurde.
  //Die mode-Konstanten (definiert in <sys/stat.h> sind:
  //S_IRUSR          user read
  //S_IWUSR          user write
  //S_IRGRP          group read
  //S_IWGRP          group write
  //S_IROTH          other read
  //S_IWOTH          other write
  //Rückgabe Filedescriptor
```

2. Nach dem ersten Open muss die Größe des Shared-Objekts bestimmt werden:

```
int ftruncate(int fd, off_t length)
```

3. Anschließend wird das Objekt in den Adressraum des Prozesses durch den Aufruf eingeblendet:

```
void * mmap(void* adr, size_t length, int prot, int flags,
          int fd, off_t offset)
  //adr: Startadresse im Prozess,wo das Objekt eingeblendet wird
  //Normalerweise ist dieser Parameter NULL.
  //length: Länge (Anzahl der Bytes, gestartet bei offset,
  //die eingeblendet werden sollen).
  //prot:      Protection oft: PROT_READ, PROT_WRITE oder
  //PROT_READ | PROT_WRITE,
  //flags MAP_SHARED | MAP_PRIVATE | MAP_FIXED, we use MAP_SHARED
  //fd: Filedescriptor des Shared-Memory
  //offset:      ist normalerweise 0.
  //return Adresse des gemappten Shared-Memory
```

Nach der Verwendung wird das Objekt mit {munmap} getrennt, mit {shm_unlink} wird der Name des Objekts gelöscht. Das Objekt wird nur gelöscht, wenn keine Referenz darauf mehr existiert.

```
int main()   //Einfaches Programm z. Anlegen eines Shm-Objekts
{
  //Versuch, ein Shared-Memory-Objekt zu erzeugen
  int fd=shm_open("TestSharedMem",O_RDWR | O_CREAT | O_EXCL,FILE_MODE);
  bool success = true;
  if (fd == -1)
  {
     if (errno != EEXIST)
     {
       sOwnCout<< "can not open " << endl;
       success = false;
     }
     else
     {
       fd = shm_open("TestSharedMem", O_RDWR, FILE_MODE);
       if (fd == -1)
       {
         sOwnCout<< "check permission " << endl;
         success = false;
       }
       else
       {
         sOwnCout<< "open an existing Shared-Memory" << endl;
       }
     }
  }
  else
  {
     sOwnCout<< "create a new Shared-Memory " << endl;
  }
  if (true == success)
  {
     ftruncate(fd, 1000);
  }
  return 0;
}
```

placement new

Anlegen eines festen Speicherpuffers im
Datensegment während der Startzeit oder zur Kompilier/Link-Zeit. Die
Objekte werden dann zu einem späteren Zeitpunkt durch Placement-new in
diesen Speicherbereich abgelegt.
STATIC_MEM_SIZE ist dabei eine Konstante aus einer config.h-Datei, die in der Entwicklungsphase in der Größe bestimmt werden muss (Buchführung).

```
static char sMem[STATIC_MEM_SIZE];      // Static Speicher für Objekte
static char* sCurrentPtr = sMem;        // Merker für freien Speicher
static const char* sMemEndPtr = sMem+STATIC_MEM_SIZE;  //Speicherende
```

Aus diesem Puffer können nun im Betrieb quasi-statische Objekte
angefordert werden.
Die Objekte werden nur einmal angelegt und existieren bis zum Herunterfahren des Programms.
Die folgende C-Funktion liefert den Zeiger auf statischen Objektspeicher der gewünschten Größe, angelegt in dem oben generierten statischen
Mit den Zeilen {sMutex.take} und {sMutex.give} werden auch hier wieder Synchronisationsmechanismen um die eigentliche Funktion gelegt, so dass Kollisionen durch konkurrierende Zugriffe vermieden werden.

```
char * getStaticMemory(size_t memSize)
{                               //gibt Zeiger auf Speicher
  sMutex.take();
  ASSERTION(sCurrentPtr + memSize < sMemEndPtr); //eig. Fehlerbehdlng
  char *retPtr = sCurrentPtr;          //ret Zeiger auf freien Speicher
  sCurrentPtr+=MAKE_ALIGNMENT_SIZE(memSize);
  sMutex.give();                    //und Weiterschaltung
  return retPtr;
}
```

Mit einem Placement-new-Operator (manchmal auch Replacement-new genannt) kann man eine Variable des gewünschten Typs im Pufferspeicher vereinbaren.

```
ObjPtr = new(getStaticMemory(sizeof(int))) int;
            //Objekt (z.B. int) im vorgegebenen Speicher anlegen

varPtr = new (location) Type(Param);      //Konstruktor mit Parameter
```

Entweder mit eigenen Placement-new-Konstrukten oder besser mit vorgefertigten Makros kann nun statischer Speicher aus dem Datensegment allokiert werden.

```
#define NEW_STATIC_DEFAULT(Typ)(new(getStaticMemory(sizeof(Typ)))Typ())
#define NEW_STATIC(Typ,ARGS)(new(getStaticMemory(sizeof(Typ)))Typ ARGS)
```

Queue

Wie schon einige Male erwähnt, sind Queues Warteschlangen. Die Warteschlange an englischen Bushaltestellen ist ein anschauliches Beispiel für deren Funktionsweise. Kommen neue Fahrgäste zur Bushaltestelle, so müssen sie sich hinten an der Warteschlange anstellen (please queue up!). Diejenigen, die ganz vorne stehen, haben bereits am längsten gewartet und dürfen zuerst in den Bus einsteigen und mitfahren. Die anderen Fahrgäste in der Schlange dürfen dann nach vorne nachrücken.

Es handelt sich also um das FIFO-Prinzip, "`first in first out"'. Es ist eine Queue mit tail-write-Eigenschaft, neue Elemente werden hinten angefügt.

```
class CCommQueue
{
public:
  CCommQueue(CBinarySemaphore& sem);
  ~CCommQueue();

  bool add(const CMessage& msg);  // add Message into the queue
  bool getMessage(CMessage& msg); // get a Message from the queue
  Int32 getNumOfMessages(void);   // get the number of Messages
  Int32 getNumOfByteNeeded(void); // get number of Bytes occupied
                                  // by the queue
private:
  Int32      mSize;        // size of the queue
  Int32      mCurrentSize;  // current size
  Int32      mHeadIndex;    // first possible element for reading
  Int32      mTailIndex;    // position to insert
  CMessage   mQueue[1024]         // too big

 };
```

Event

Events werden als Kommunikationsmittel zwischen Komponenten eingesetzt. Ob die Kommunikation synchron oder asynchron ist, hängt von der konkreten Anwendung ab. In der Regel ist eine Anfrage synchron. Die Benachrichtigung, wenn bestimmte Ereignisse eintreffen, ist naturgemäß asynchron, da zeitlich nicht voraussagbar.
Für Events werden Objekte fester Größe verwendet. Wir konzentrieren uns auf die Kontrollnachrichten. Kleine Datenmenge können damit transportiert werden. Größere Datenmengen werden über einen gemeinsamen Speicherbereich abgewickelt.

Events können auf vielerlei Arten realisiert werden.
Einfache Events werden mit einem Enum-Typ implementiert und an einer zentralen Stelle, die jedem Nutzer zugänglich ist, wird die Bedeutung jeder Event-Nummer festgehalten. Wie in der Literatur werden hier die Begriffe Event und Message synonym verwendet.

```
enum MessageType
{
  TUNER_SCAN,
  TUNER_SETFREQUENCY,
  AMP_SETVOL,
  AMP_SETMUTE,
  ...
};
...
queue.add(TUNER_SCAN) ;   //Enum-Wert wird in die Queue eingetragen
```

```
  MessageType event;
  queue.get(event) ;
  switch (event)
  {
  case TUNER_SCAN:
    mTuner.scan();
    break;
  case TUNER_SETFREQUENCY:
    mTuner.setFrequency();
    break;
  ...
  default:
    break;
  }
```

Eine weitere einfache Version ist die C-Struktur, in der Einträge über ihre Elementnamen angesprochen werden können:

```
struct MyMessage
{
  UInt8   mType;
  UInt8   mData[63];
};
```

MOST-Messages für Kommandos oder für Daten gemäß der Spezifikation \cite{MOST} sind ebenfalls einfache Events:

```
struct MOSTEvent
{
  UInt16   mDeviceID;
  UInt8    mFBlockID;
  UInt8    mInstance;
  UInt16   mFkID:12;   // Bitfeld
  UInt16   mOPType:4;
  UInt8    mLength;    // bis hierhin sind es 7 Bytes
  UInt8    mData[12];  // hier sind die Nutzdaten
} ;
```

Auf die einzelnen Bedeutungen gehen wir später ein. Ersichtlich sind solche Events nach serialisierter Übertragung typfrei, d.h., zwischen Sender und Empfänger muss ein gemeinsames Format, das nicht Teil der Übertragung ist, vereinbart sein. Dies ist z.B. das oben angegebene Format, das beide, der Sender und der Empfänger aus einer Headerdatei kennen.

```
class CMessage;                                    // forward declaration
class IMessageHandler
{
 public:
   virtual void handleMessage(const CMessage& msg) = 0;
};
```

Die Bestandteile von CMessage:

```
  class Header
  {
   public:
     Int32 mSenderID;                              //die Kennung des Senders
     Int32 mReceiverID;                            //Kennung des Empfängers
     IMessageHandler* mMessageHandlerPtr;          //Nachrichten-Handler
  };
```

Die Komponenten haben zusätzlich zur {FBlockID} eine eigene ID.
Dies ist nötig, da manche Komponenten nur
intern verwendet werden und keinen Bezug zu MOST-Function-Blöcken haben.

```
  class MostContent
  {
   public:
     Int8 mType;                                   // tick, admin, poolref
     Int8 mSource;                                 // extern, intern
     Int16 mLength;                                // länge der nachricht
     Int8 mMostMessage [MESSAGE_SIZE - sizeof(Header) - 4];       //nutzdaten
  };

  class NormalContent
  {
   public:
     Int8 mType;
     Int8 mReserved [3];                       // wegen padding
     Int32 mOpcode;
     Int32 mParam[(MESSAGE_SIZE - sizeof(Header) - 8)/sizeof(Int32)];
  };
```

```
class CMessage
{
 public:
   enum MessageType
     {
       ADMIN_TYPE,                 // for admin messages
       TICK_SERVICE_TYPE,      // for tick event
       ...
       KEY_EVENT_TYPE,          // for key event
     };
   enum SourceType
   {
     EXTERNAL_TYPE = 0,
     DEVICE_TYPE = 2,
     MODEL_TYPE = 3
   };

   class Header
   {
     ...// see above
   };
   class NormalContent
   {
     ... // see above
   };
   class MostContent
   {
     ... // see above
   };

   CMessage();
   CMessage(MessageType theMsgType);

   // SETTER-METHODS
   void setSenderID(Int32 senderID);
   void setReceiverID(Int32 receiverID);
   void setOpcode(Int32 opcode);
   void setParam(UInt32 index, Int32 value);
   void setMessageHandlerPtr(IMessageHandler* handlerPtr);
   void setSource(SourceType type);

   // GETTER-METHODS
   Int32  getSenderID(void) const;
   Int32  getReceiverID(void) const;
   MessageType getMessageType(void) const;
   Int32  getOpcode(void) const;
   Int32  getParam(UInt32 index) const;
   UInt32 getParamCount(void) const;
   IMessageHandler* getMessageHandlerPtr(void) const;

   Int8 * getMostContentPtr(void) const;
   UInt32 getMostContentSize(void) const;
   UInt32 getMostMessageSize(void) const;
   SourceType getSource(void) const;

private:
   Header          mHeader;
   union                                  // and one of the following contents (depending on message type)
   {
     MostContent    mMostContent;
     NormalContent  mNormalContent;
   };
```

```
};
```

```
void CProxy::doIt()
{
  CMessage msg(CMessage::REQUEST_TYPE);
  setSenderID(mID);
  setReceiverID(mServiceID);
  msg.setOpcode(Demo::DO_IT);
  // mQueuePtr ist die Queue der betreffenden Komponenten
  mQueuePtr->add(msg);
}

void CProxy::connect(UInt32 channels)
{
  CMessage msg(CMessage::REQUEST_TYPE);
  setSenderID(mID);
  setReceiverID(mServiceID);
  msg.setOpcode(Demo::CONNECT);
  msg.setParam(0, channels);
  // mQueuePtr ist die Queue der betreffenden Komponenten
  mQueuePtr->add(msg);
}
```

Auf der Empfängerseite wird eine Handler-Klasse implementiert. Die Handler-Klasse kennt den Dienstanbieter und ist bei dem Dispatcher registriert (siehe unten), so dass er die Nachricht bekommt. Seine handle-Message-Methode sieht dann etwa so aus:

```
void CHandler::handleMessage(const CMessage& msg)
{
  if (msg.getOpcode() == Demo::DO_IT)
    mServiceProvider.doIt();
  else
  {
    UInt32 param0 = static_cast<UInt32>(msg.getParam(0));
    mServiceProvider.connect(param0);
  }
}
```

Dispatcher

Der System-Dispatcher muss die Briefkästen aller Komponenten kennen. In diese verteilt er eingegangene Nachrichten. In unserer Implementierung befinden sich alle Queues der Komponenten im Shared-Memory. Der System-Dispatcher kann sich von allen Komponenten über den jeweiligen Kontext die {add}-Methoden der jeweiligen Queues holen.
Damit kann der Dispatcher alle Queues ansprechen und sie abhängig von den im Event-Objekt gespeicherten Informationen füllen.

Tatsächlich kann der System-Dispatcher sowohl für das Verteilen eingegangener Events als auch für das Aussenden im System generierter Events verantwortlich sein.

Der System-Dispatcher empfängt aus drei Quellen Events, die in zwei möglichen Formaten vorliegen können und in der Headunit verteilt werden müssen. Alle Events werden in einer Queue aufgefangen.

- Vom externen MOST-Bus werden MOST-Events empfangen und müssen an eine der internen Komponenten weiterverteilt werden.
- Von einem Tastaturtreiber werden Daten abgeholt, die an die HMI-Komponente weitergeleitet werden müssen. Dort können sie dann im Zusammenhang mit dem aktuellen Menü ausgewertet werden.
- Von internen Komponenten empfängt er Events, die intern oder nach außen weiterverteilt werden müssen. Diese Events werden in derselben externen Queue wie oben aufgesammelt
- Eine interne Komponente sendet Nachrichten über den Dispatcher an externe MOST-Devices.
- Eine interne Komponente sendet Nachrichten über den Dispatcher an interne MOST-Devices.

Admin

In der Admin-Komponente werden alle Komponenten mit deren Kontexten wie Shared-Memories und Queues angelegt. Ggf. werden auch watchdogs und timer implementiert. Näheres siehe Buch.

Das System und das Framework

Ein- und Ausgangssignale werden schon in einer Treiberschicht in einheitliche logische Signale (Events) verwandelt. Jeder Treiber für eine physikalische Einheit ist ein eigener Task (Bedieneinheit, Grafikdisplay, Laufwerk, Festplatte, ...).
Wir folgen der Abstraktion der MOST-Spezifikation und definieren in der Regel für jeden Function-Block (MOST-Gerät) in der Headunit eine Komponente, in der das Modell abläuft.
Wir definieren eine HMI-Komponente, in der die Bedienabläufe aller Devices implementiert sind, da sie stark miteinander verwoben und nicht gut trennbar sind. Im MVC-Sprachgebrauch sind das die Controller.
Auch die Implementierung der virtuellen GUI mit abstrahierten Bedienelementen und Ausgaben (GUI-Objekte) findet sich in der HMI-Komponente. Im MVC-Gebrauch ist dies der View.
Die eigentliche Grafikerzeugung wird dem Treiber oder einem separaten Task zugerechnet.
Weiterhin wird es einige zusätzliche Tasks wie Admin oder Komponenten wie Audio-Controller usw. geben.
Eine weitere Komponente ist der System-Dispatcher, der die Events nach innen und außen verteilt
Um externe Events zu bearbeiten, verfügt jede Komponente
mindestens über eine Queue und einen Dispatcher.
Im Folgenden sprechen wir von Komponenten oder von Tasks, wobei Komponenten Tasks mit zusätzlichen Vorrichtungen wie Queues, Timer und Watchdog sind.