Echtzeitbetriebssysteme (am Beispiel QNX)

Dr. Stefan Enderle

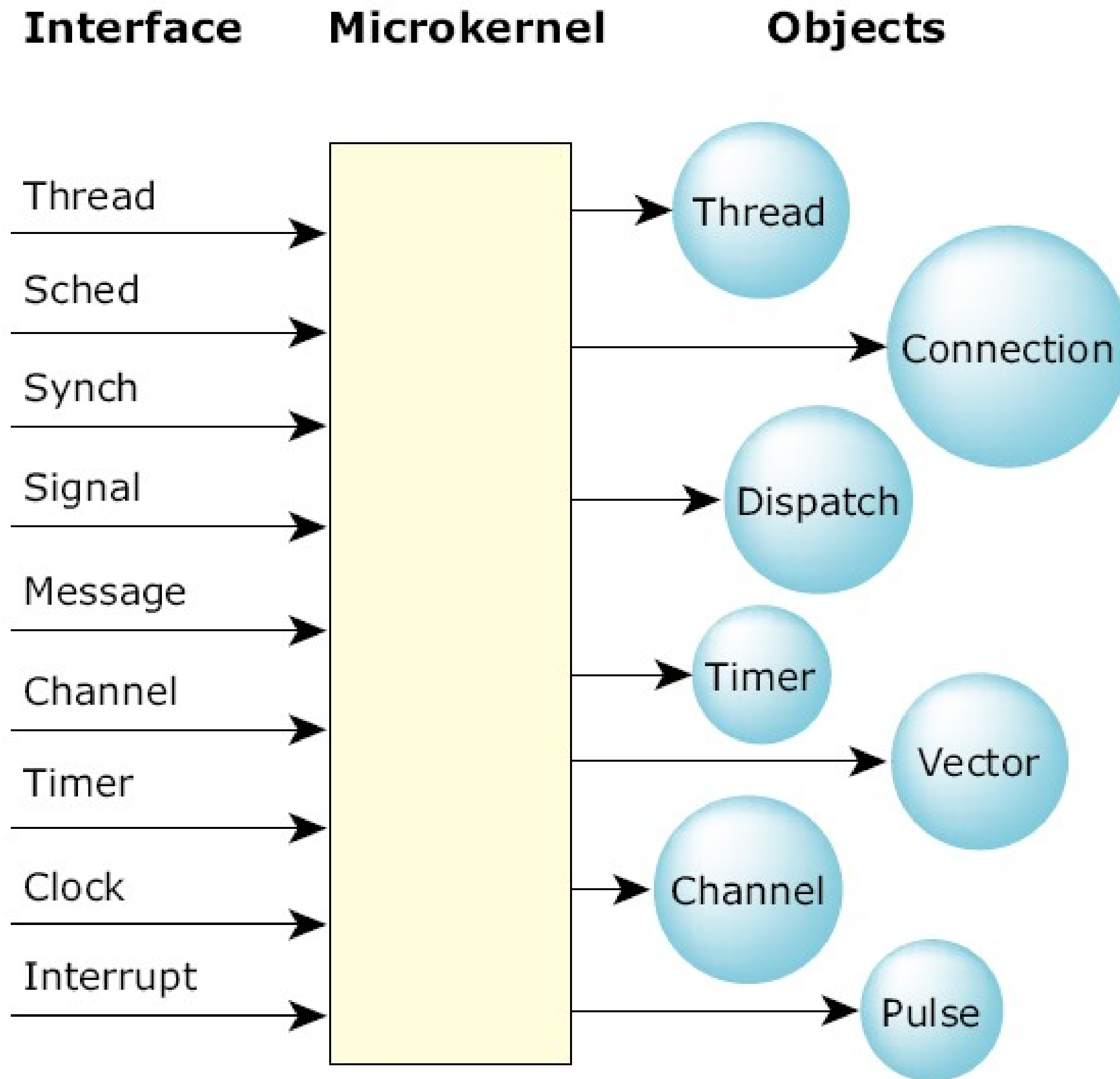
HS Esslingen

3. QNX – Neutrino Microkernel

3.1 Einführung

- Der **Neutrino-Kernel** (Betriebssystemkern) implementiert (nur) die wichtigsten Teile des Betriebssystems.
- **"Microkernel"**
- Features, die nicht im den Kernel realisiert sind, laufen in **normalen Prozessen** ab (z.B. File-I/O, Device-I/O)
- Auf der untersten Ebene werden nur wenige **Objekt-Arten** unterstützt, diese aber sehr performant!

Kernel-Objekte



System Services

- Neutrino bietet **Kernel-Aufrufe** für folgende Objekte:
  - threads
  - message passing
  - signals
  - clocks
  - timers
  - interrupt handlers
  - semaphores
  - mutual exclusion locks (mutexes)
  - condition variables (condvars)
  - barriers
- Das gesamte Betriebssystem baut auf diesen Aufrufen auf.

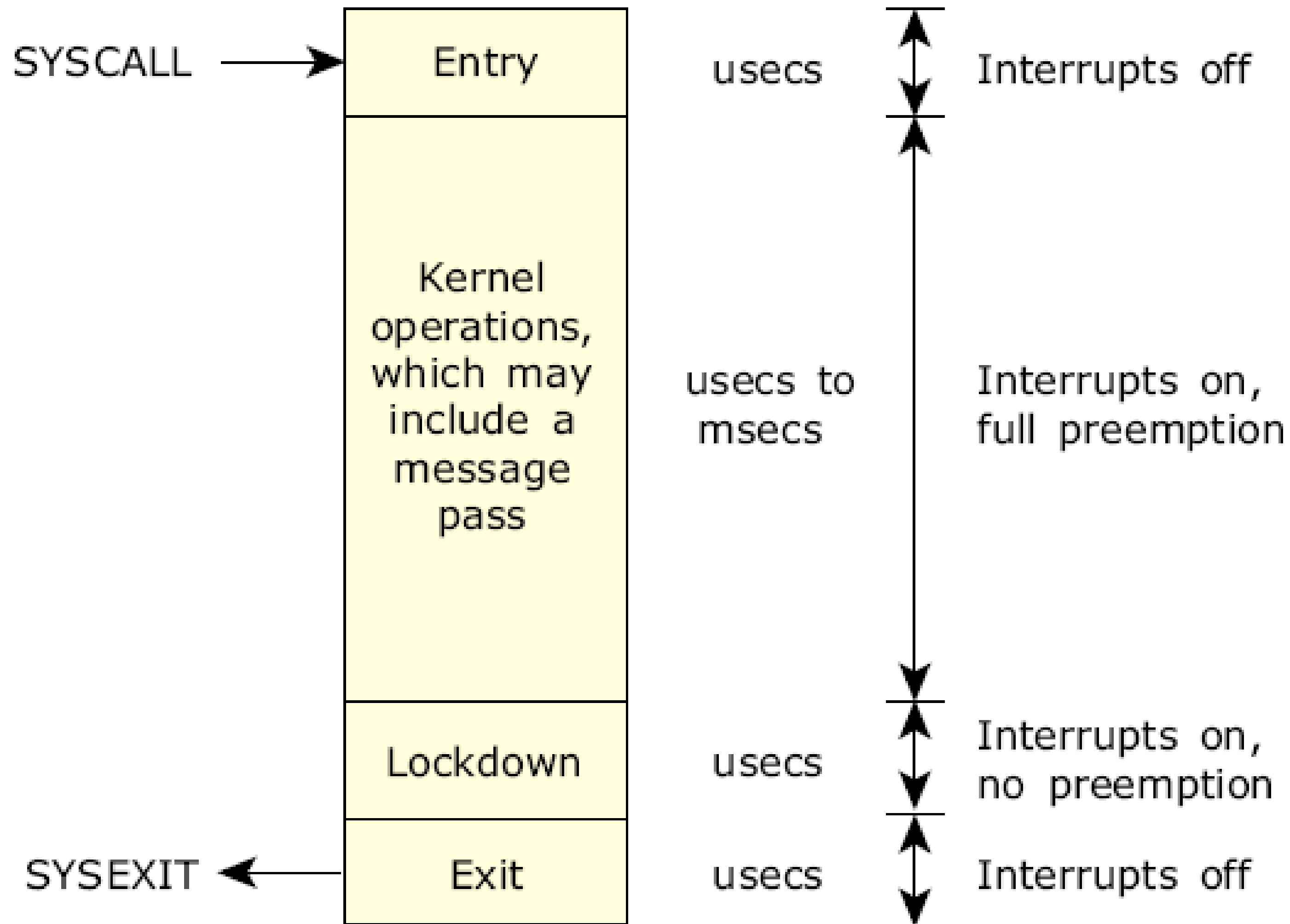
System Calls

- Der Aufruf eines bestimmten System Services wird als **System Call** bezeichnet.

- In QNX sind System Calls **unterbrechbar** durch andere Prozesse mit höherer Priorität. (Letztlich unterbricht aber der Kernel!)

- Es gibt nur kurze Bereiche (< 1ns) zu Beginn und Ende eines System Calls, zu denen nicht unterbrochen werden kann. (Interrupts Off)

Preemptive System Calls



3.2 Threads (Wdh.)

- Threads sind "leichtgewichtige" Prozesse, die vom Betriebssystem schnell zu starten und zu stoppen sind.

- Threads laufen parallel zum main-Prozess.

- Als Thread können Funktionen mit folgender Signatur gestartet werden:

  ```
  void* <funktionsname> (void* <args>)
  ```

  Wir verwenden meist:

  ```
  void* <funktionsname> (void* not_used)
  ```

Thread Funktionen

- **Wichtige Thread-Funktionen:**

- ***pthread_create()*** Create a new thread of execution.
- ***pthread_exit()*** Destroy a thread.
- ***pthread_detach()*** Detach a thread so it doesn't need to be joined.
- ***pthread_join()*** Join a thread waiting for its exit status.
- ***pthread_cancel()*** Cancel a thread at the next cancellation point.

Thread erzeugen

```
#include <stdio.h>


void* thread1 (void* not_used)
{
  printf("thread1 running\n");
}


int main ()
{
  pthread_create (NULL,NULL,thread1,NULL);
  sleep(1);
}
```

Parallele Threads

```
#include <stdio.h>

void* thread1 (void* not_used)
{
  while(1) {
    printf("thread1 running\n");
    sleep(1);
  }
}

void* thread2 (void* not_used)
{
  while(1) {
    printf("thread2 running\n");
    sleep(3);
  }
}

int main ()
{
  pthread_create (NULL,NULL,thread1,NULL);
  pthread_create (NULL,NULL,thread2,NULL);
  sleep(15);
}
```

Aufgabe: Viele Threads

- Versuchen Sie 100 Threads mit der gleichen Funktion zu erzeugen.

Thread ID

- Jeder mit `pthread_create()` angelegte Thread bekommt vom Betriebssystem eine **eindeutige ID**.
- Über diese ID kann man von anderen Threads aus auf einen Thread zugreifen.
- **Abfrage der ID:**

```
#include <pthread.h>

pthread_t id;
pthread_create(&id, NULL, thread, NULL);
```

Aufgabe: Thread ID

- Einige Threads erzeugen, die pro Sekunde etwas ausgeben.
- Die IDs der erzeugten Threads ausgeben.

Thread ID (2)

- Ein Thread kann seine eigene Thread ID abfragen.

- **Abfrage der eigenen ID:**

```
void* thread1 (void* not_used)
{
  int id;
  id = pthread_self();
}
```

Aufgabe: Thread ID

- Einige Threads erzeugen, die pro Sekunde ihre eigene ID ausgibt.
- Die IDs der erzeugten Threads ausgeben.
- Die IDs vergleichen...

Thread selbst beenden

- Ein Thread kann sich selbst beenden.
- (Entspricht aber dem normalen Verlassen der Funktion.)

- **Thread selbst beenden:**

```
void* thread1 (void* not_used)
{
   ...
   pthread_exit(NULL);
   ...
}
```

Aufgabe: exit

- Einige Threads erzeugen, die pro Sekunde ihre ID ausgeben und sich nach 2*ID Sekunden beenden.
- Die IDs der erzeugten Threads ausgeben.
- Warten...

Beenden eines Threads

- Über die Thread ID kann ein Thread durch einen anderen Thread beendet werden.
- Cleanup-Funktionen werden ausgeführt (s.u.)

- **Thread beenden:**

```
pthread_t id;
pthread_create(&id, NULL, thread, NULL);

pthread_cancel(id);
```

Aufgabe: cancel

- Einen Thread erzeugen, der pro Sekunde etwas ausgibt.
- Die ID des erzeugten Threads ausgeben.
- Den Thread nach 3 Sekunden beenden.
- Im Hauptprogramm weitere 3 Sekunden warten.

Cleanup Funktionen

- Ein Thread kann eine "Cleanup"-Funktion definieren, die aufgerufen wird, wenn der Thread beendet wird (exit oder cancel).

- **Cleanup-Funktion einhängen:**

```
void cleanup_fct (void* not_used)
{
   ...
}

void* thread1 (void* not_used)
{
  pthread_cleanup_push(cleanup_fct, NULL);
   ...
}}  <-- Internes Problem!!
```

Aufgabe: cleanup

- Eine Cleanup-Funktion definieren, die "Cleanup" ausgibt.
- Einen Thread erzeugen, der die Cleanup-Funktion einhängt und dann pro Sekunde etwas ausgibt.
- Den Thread nach 3 Sekunden per **`cancel`** beenden.
- Den Thread neu starten
- Den Thread nach 3 Sekunden per **`abort`** beenden.
- Im Hauptprogramm weitere 3 Sekunden warten.

Hartes Beenden eines Threads

- Über die Thread ID kann ein Thread sofort beendet ("abgeschossen") werden.
- **Achtung:**
  Cleanup-Funktionen werden nicht mehr ausgeführt (s.u.)
- **Thread sofort beenden:**

```
pthread_t id;
pthread_create(&id, NULL, thread, NULL);

pthread_abort(id);
```

Aufgabe: abort

- Einen Thread erzeugen, der pro Sekunde etwas ausgibt.
- Die ID des erzeugten Threads ausgeben.
- Den Thread nach 3 Sekunden hart beenden.
- Im Hauptprogramm weitere 3 Sekunden warten.

Canceln eines Threads verhindern

- Ein Thread kann verhindern, dass er gecancelt wird.
- (Er kann dies auch wieder zulassen.)

- **Cancelstate ändern:**

```
void* thread1 (void* not_used)
{
  pthread_setcancelstate
          (PTHREAD_CANCEL_DISABLE, NULL);
  ...
  pthread_setcancelstate
          (PTHREAD_CANCEL_ENABLE, NULL);
}
```

Aufgabe: cancelstate ändern

- Eine Cleanup-Funktion definieren, die "Cleanup" ausgibt.
- Einen Thread 1 erzeugen, der die Cleanup-Funktion einhängt und dann pro Sekunde etwas ausgibt.
- Einen Thread 2 erzeugen, der den "Cancelstate disabled", die Cleanup-Funktion einhängt und dann pro Sekunde etwas ausgibt.
- Beide Threads nach 3 Sekunden per **`cancel`** beenden.
- Beide Threads neu starten
- Beide Thread nach 3 Sekunden per **`abort`** beenden.

"Joinen" eines Threads

- Ein Thread kann sich an einen anderen Thread anhängen.
- Der aktuelle Thread blockiert dann und läuft erst weiter, wenn der "gejointe" Thread beendet wird.
- **Thread join:**

```
pthread_t id;
pthread_create(&id, NULL, thread, NULL);

pthread_join(id, NULL);
```

Aufgabe: join

- Einen Thread erzeugen, der
  - etwas ausgibt,
  - 3 Sek. wartet,
  - noch etwas ausgibt,
  - sich dann beendet.
- Den Thread joinen
- Danach etwas ausgeben

"Joinen" eines Threads (2)

- Ein Thread will einen anderen Thread joinen.
- **Frage:** Woher kann ein Thread die ID eines anderen Threads wissen?
  1. Er erzeugt den Thread selbst
  2. Das Hauptprogramm übergibt ihm die ID
     1. Als global Variable ← Pfui!
     2. Gleich beim Aufruf von **`pthread_create()`**

- **Bisher:**

```
pthread_t id;
pthread_create(&id, NULL, thread, NULL);
```

Parameterübergabe bei create

- Das letzte Argument bei **`pthread_create`** ist eigentlich ein Zeiger auf void: **`void*`**
- Beim Aufruf kann hier ein beliebiger Zeiger übergeben werden.
- In der Thread-Funktion muss dieser Zeiger dann "ge-typecasted" werden:

```
int data;
pthread_t id;
pthread_create(&id, NULL, thread1, &data);

void* thread1 (void* data_ptr)
{
  int data = *((int*) data_ptr);
}
```

Aufgabe: join (2)

- Einen Thread erzeugen, der
  - etwas ausgibt,
  - 3 Sek. wartet,
  - noch etwas ausgibt,
  - sich dann beendet.
- Einen zweiten Thread erzeugen, der
  - die übergebene ID des ersten Threads ausgibt,
  - diesen Thread dann joined,
  - danach noch etwas ausgibt.
- Im Hauptprogramm die beiden Threads erzeugen und die IDs ausgeben, dem zweiten Thread die ID des ersten übergeben und den zweiten Thread joinen.

Thread-Attribute

- **Jeder Thread besitzt einige private Daten:**
  - **thread ID** Each thread is identified by an integer thread ID, starting at 1. The ID is unique within the thread's process.
  - **register set** Each thread has its own instruction pointer (IP), stack pointer (SP), and other processor-specific register context.
  - **stack** Each thread executes on its own stack, stored within the address space of its process.
  - **signal mask** Each thread has its own signal mask.
  - **thread local storage** A thread has a system-defined data area called "thread local storage" (TLS). The TLS is used to store "per-thread" information (such as *tid*, *pid*, stack base, *errno*, and thread-specific key/data bindings). The TLS doesn't need to be accessed directly by a user application. A thread can have user-defined data associated with a thread-specific data key.
  - **cancellation handlers** Callback functions that are executed when the thread terminates.

Thread-Attribute

- Beim Erzeugen eines Threads mit **`pthread_create()`** können einige dieser Attribute speziell gesetzt werden.

- **Beispiel: Nicht-joinable Thread erzeugen:**

```
int main ()
{
  pthread_attr_t attrib;

  pthread_attr_init(&attrib);
  pthread_attr_setdetachstate(&attrib,
                        PTHREAD_CREATE_DETACHED);

  pthread_create (NULL,&attrib,thread1,NULL);
}
```