Real-Time Programming

Lecture 2

Farhang Nemati

Spring 2016

Properties of Real-Time Systems

- Complexity
  - Can vary from a small system with a few lines of code to several millions of code, e.g., automotive systems
- Reliability
  - Real-Time systems usually control some components in the physical world where a failure may have severe or catastrophic consequences, e.g., flight control system. They have to be reliable!
- Concurrency
  - Several things are done in parallel. The system is splitted into multiple tasks run in parallel.

Properties of Real-Time Systems

- Interactive with physical world
  - The system interacts with environment through sensors and actuators and special purpose hardware.
- Schedulabilty
  - The tasks have to finish their work in time. Each real-time task has a deadline, if all tasks could run in a way that none of them misses its deadline the system is schedulable. In hard real-time systems the schedulability of the system has to be guaranteed in advance.

Properties of Real-Time Systems

- Fault Tolerant
  - In the case of a fault the system should continue working, e.g., in a safe mode
- Predictability
  - The behavior of the system has to be known and deterministic in advance, i.e., before it is deployed on a real system.

Why Predictability is difficult to achieve?

- External events: Handling interrupts may not be deterministic; we don't know when the external events happen.
- Cache problems: knowing exact cache misses/hits in advance is not possible.
- Dynamic memory allocation methods are not deterministic. Static memory may not be sufficient.

Why Predictability is difficult to achieve?

- System calls are not deterministic.
- Priority inversion: a lower priority task runs while a higher priority task is waiting.
- To guarantee the schedulability of a real-time system the Worst Case Execution Time (WCET) of each task has to be known which is very difficult.

Why Concurrency?

- Distribute work among multiple tasks
- React to external events in time
- Decompose a system into functions that can run in parallel with each other.
- Run different operations independently
  - E.g., to be able to react to a sensor input in time, sensor acquisition has to be performed independently from other tasks.

Challenges with Concurrency

- Task communication
- Tasks may share resources/data
  - Mutually exclusive resources/data has to be protected to remain consistant.
- Developing concurrent programs is hard
- Debugging concurrent programs is much harder

Types of Task communication

- External communication
  - Interaction with environment, e.g., through sensors and actuators
- Synchronization on mutual exclusive resources
- Synchronization; task may wait for a condition to be fulfilled
- Data communication; tasks may send/receive data to/from other tasks.

Timing facilities that a RTOS has to provide

- Access to Clock, i.e., system time
  - A timer interrupts the processor in constant time intervals; jiffy. Each timer interrupt is called a tick
  - The clock rate (the number of ticks per second)
    - In Linux 2.4 a tick is 10ms (100 ticks/s; 100HZ) and in Linux 2.6 it is 1ms (1000 ticks/s; 1000HZ- 1kHZ)
  - Any timing parameter of a task, e.g., deadline, has to be a multiplication of system tick to have exact precision
- Possibility for delaying the execution of a task for an interval of time

POSIX Standard

- POSIX = Portable Operating System Interface (for Unix)s
- A family of related standards
  - Provides standard Application Programming Interfaces (APIs) and command line shells and utilities for an operating system
  - Specified by IEEE
  - Facilitates developing applications portable to any operating system compatible with POSIX

POSIX Threads (PThreads)

- To work with threads (tasks) independently of language and operating system
- A thread has relatively many attributes
- The attributes of a thread are added to an attributes object (pthread_attr_t). The object is then used (passed as a parameter) when creating the thread.
  - For each attribute in the attributes object there are functions to get/set them

POSIX Threads (PThreads)

```
#include <pthread.h>
```

- Create a Pthread

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

- Wait for a Pthread to terminate

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

- Cancel a Pthread

```
int pthread_cancel(pthread_t thread);
```

POSIX Threads (PThreads)

- Example; create a Pthread

```
void *start_function( void *funcArgs )
{
  struct *myArgs = funcArgs;
    ...
}
struct threadArgs
{
   float d1;
   int d2;
   char *d3;
   ...
}
void mainFunc()
{
    struct threadArgs args;
    args.d1 = ...;
    ...
    pthread_t threadId;
    int pthread_create(&threadId, NULL/*default attributes*/, start_function, (void *)&args);
}
```

POSIX Threads (PThreads)

- Example; create a Pthread with attributes object

```
pthread_t threadId;

pthread_attr_t atr;
pthread_attr_init(&attr);
pthread_attr_setstacksize(&attr, 4000);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
...

int pthread_create(&threadId, &atr, start_function, (void *)&args);
```

POSIX Threads (PThreads)

- Get the Priority of a Pthread

```
int pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);
policy: SCHED_FIFO, SCHED_RR, SCHED_OTHER
Priority: param.sched_priority
```

- Set the Priority of a Pthread

```
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);
```

  - SCHED_FIFO, SCHED_RR are Real-Time policies. The priority is only set with these policies. With other policies the priority has to be 0.

Delay a PThread

- sleep()
  - The execution of the calling task (thread) is delayed for at least **sec** seconds

```
#include <unistd.h>

unsigned sleep(unsigned sec)
```

Delay a PThread

- nanosleep()
  - Delays the thread either for at least the duration of time specified in *rqtp or until the delay is interrupted.

```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
/*the time interval for which the task sleeps is put in rqtp*/
```

- The structure *timespec* is used to specify intervals of time with nanosecond precision:

```
struct timespec {
    time_t tv_sec;        /* seconds */
    long   tv_nsec;       /* nanoseconds */
};
```

Car Control Example, Using PThreads

```
#include <pthread.h>
#include <time.h>

void *controlEngine(void *args)
{
    struct timespec ts1, ts2;
    ts1.tv_sec = 0;
    ts1.tv_nsec = 6000000; /*6 milliseconds*/
    while(1){
        /*controlEngineOperations here. We assume it takes 4 ms */
        nanosleep(&ts1, &ts2);
    }
}
```

Car Control Example, Using PThreads

```
void *controlSpeed(void *args)
{
    struct timespec ts1, ts2;
    ts1.tv_sec = 0;
    ts1.tv_nsec = 6000000; /*6 milliseconds*/
    while(1){
        /*controlSpeedOperations here. We assume it takes 4 ms */
        nanosleep(&ts1, &ts2);
    }
}
void *controlFuel(void *args)
{
    struct timespec ts1, ts2;
    ts1.tv_sec = 0;
    ts1.tv_nsec = 18000000; /*18 milliseconds*/
    while(1){
        /*controlFuelOperations here. We assume it takes 2 ms */
        nanosleep(&ts1, &ts2);
    }
}
```

Car Control Example, RTOS Solution

```
int main()
{
    pthread_t engine, speed, fuel;
    struct sched_param eparam, sparam, feparam;
    eparam.sched_priority = 10;
    separam.sched_priority = 11;
    feparam.sched_priority = 12;

    pthread_create(&engine, NULL, controlEngine, NULL);
    pthread_create(&speed, NULL, controlSpeed, NULL);
    pthread_create(&fuel, NULL, controlFuel, NULL);

    pthread_setschedparam(engine, SCHED_FIFO, &eparam);
    pthread_setschedparam(speed, SCHED_FIFO, &sparam);
    pthread_setschedparam(fuel, SCHED_FIFO, &fparam);

    pthread_join(engine, NULL);
    pthread_join(speed, NULL);
    pthread_join(fuel, NULL);
}
```