Lecture 12: POSIX threads programming (Burns & Wellings ch. 7-8)

Posix threads – a crash course

- thread concept, CPU, OS
- creating threads
- terminating threads
- setting thread priorities
- simple delaying
- inter-process communication

Reactive objects and threads

- Thread
  - basic execution entity
- Reactive object
  - object with (possibly) its own thread
- Traditionally
  - threads have been used when programming
  - local variables for the thread
  - essentially makes it an object
  - if variables are not accessed elsewhere
- Today: threads
  - with posix

A CPU seen from the outside

- The program = a set of code fragments that constitute the reactions to recognized events
- Input detection = the invocation of such a fragment (a function call)

The reactive embedded system

Application

Sensor

Port

Port

Sensor

Port

Port

Actuator

A CPU seen from the inside

- The program = a set of threads of control that run from start to stop (or infinitely), reading and writing data as they go
- Input detection = the exit from a busy-waiting fragment (a function return)
The "standard" embedded system

POSIX

A standardized operating system interface and environment, including:
- system calls
- standard C libraries
- a command shell

Based on various flavors of Unix, but vendor-independent

Modularized set of standards for
- File operations, devices
- Process management, signals
- Threads (pthreads)
- Real-time signals and scheduling

No standard for interrupts and drivers

Creating threads

```c
#include <pthread.h>

pthread_t id;

void* fun(void* arg) {
    // Some code sequence
}

main() {
    pthread_create(&id, NULL, fun, NULL);
    // Some other code sequence
}
```

Thread parameterization

```c
#include <pthread.h>

pthread_t id;

void* fun(void* arg) {
    int x = (int)arg;
    // Some code sequence
}

main() {
    pthread_create(&id, NULL, fun, (void*)7);
    // Some other code sequence
}
```

warning! void* and int may be of different size - usually not - check for each platform

Create a thread with priority 12 (default is 10, higher values mean "more important"):

```c
#include <sched.h>

int x = ...;

int thread_t;
int info = sched_param param;
pthread_attr_t attr;
pthread_attr_init(&attr);
// init attributes
pthread_attr_setinheritsched(attr, PTHREAD_EXPLICIT_SCHED);

// Set priority
param.sched_priority = 12;

pthread_create(&thread_t, &attr, param, &x);
// set parameters in attr structure
pthread_create(&thread_t, attr, fun, argy);
// create the thread

Note: priority is inherited from parent unless PTHREAD_EXPLICIT_SCHED is set

Thread termination

```c
pthread_create(&id,...)
pthread_create(&id,...)
pthread_create(&id,...)
pthread_create(&id,...)
```

joining

Non-termination

May be used to pass a return value (see manual pages)
More on termination

- Terminates a thread:
  `pthread_exit(NULL);`

- Terminates the whole process (with all its threads):
  `exit(0);`

- Returning from `main()` means (equivalent)
  `exit(return_value);`

- Returning from the function given to `create()` means
  `pthread_exit(return_value);`

**NOTE:** `main()` must be kept from terminating
all threads will terminate with `main`
use `join` (or do real work, ...)

The old sonar example

```c
int dist, sig;
while(1) {
  sonar_read(&dist);
  pthread_mutex_lock(&mu);
  control(dist, &sig, &params);
  pthread_mutex_unlock(&mu);
  servo_write(sig);
}
```

The sonar with mutexes

```c
struct Params params;
```

```c
void *controller(void *arg) {
  int dist, sig;
  while(1) {
    sonar_read(&dist);
    pthread_mutex_lock(&mu);
    control(dist, &sig, &params);
    pthread_mutex_unlock(&mu);
    servo_write(sig);
  }
}
```

```c
struct Params params;
```

Accessing ports

- Use the standard file operations (`open`, `read`, `write`, `close`) on device files; these are typically located in the directory `/dev`
- For example: the COM1 port is named `/dev/ttyS0`
- See man page `termios` to find out about ways to configure individual port settings like baud rate, etc

Clock services

- Reading the real-time clock:
  ```c
  #include <time.h>
  struct timespec t;
  clock_gettime(CLOCK_REALTIME, &t);
  printf("%d sec, %d nanosec", t.tv_sec, t.tv_nsec);
  printf(" since Jan 1, 1970\n");      // EPOCH
  ```

- Reading number of ticks:
  ```c
  printf("%d ticks since program start\n", clock());
  ```

Notes on ticks

- Most other operating systems are set up to receive a regular clock interrupt with a period in the millisecond range – the system tick
- With some operating systems, the tick length is fixed, with others it might be configurable — use
  ```c
  clock_getres(CLOCK_REALTIME, &t)
  ```
  to find out the clock granularity
More on POSIX clocks

- Setting the real-time clock:
  ```c
  struct timespec t;
  t.tv_sec = 123456789;  // seconds since Jan 1 1970
  t.tv_nsec = 0;
  clock_settime(CLOCK_REALTIME, &t);
  ```
- Note that setting the time can cause negative time leaps. To read a clock that is always increasing, use 
  `CLOCK_MONOTONIC` // can’t be set 
- For completeness, there’s a third kind as well: 
  `CLOCK_SOFTTIME` // same as REALTIME, but stops during system sleep

Simple delaying

- Sleeping for seconds:
  ```c
  sleep(123);
  ```
- Sleeping for milliseconds:
  ```c
  delay(456);
  ```
- Sleeping for nanoseconds:
  ```c
  struct timespec t;
  t.tv_sec = 0;
  t.tv_nsec = 789;
  nanosleep(&t, NULL);
  ```
- All these services use the time of call as the time-base for the given time offset