Real-Time Operating Systems

ROS01

Minor Embedded Systems

Week 5
Using TI-RTOS

versd@hr.nl
brojz@hr.nl

Planning ROS01

- Week 1: Introduction – Blinking leds
- Week 2: Super loop construct with an ISR
- Week 3: Cooperative Scheduling
- Week 4: Pre-emptive Scheduling
- **Week 5: Using TI-RTOS**
- Week 6: Schedulability Analyses, Priority Assignment
- Week 7: Response Time Analyses
- Week 8: Finalizing Final Assigment

Overview

- Tasks
  - Creation / Deletion
  - Parameter passing
- Multitasking problems
  - Situation / Problem
  - Goal
  - Solution

Multitasking

"An environment where program execution can be interrupted and continued at any time in any location"

- Questions
  - How to design such a system and promise timing?
  - How to prevent data corruption?
  - How to communicate between tasks?

POSIX

POSIX

- Portable Operating System Interface (POSIX) is a standard API for Operating Systems.
  – Many OS partially comply with this standard. For example: Linux, Android, OSX, VxWorks, QNX Neutrino, TI-RTOS etc.
- Tasks (threads) are dynamically created by using API calls.
- Semaphores and mutexes can be used to synchronize tasks.
- Message Queues can be used to communicate between tasks.

Pthread Example (1 of 2)

```
void *print1(void *par) {
    for (int i = 0; i < 10; i++) {
        usleep(100000);
        printf("print1\n");
    }
    return NULL;
}

void *print2(void *par) {
    for (int i = 0; i < 10; i++) {
        usleep(200000);
        printf("print2\n");
    }
    return NULL;
}
```

Pthread Example (2 of 2)

```
void *main_thread(void *arg) {
    pthread_attr_t pta;
    pthread_attr_init(&pta);
    pthread_attr_setstacksize(&pta, 1024);

    pthread_t t1, t2;
    pthread_create(&t1, &pta, &print1, NULL);
    pthread_create(&t2, &pta, &print2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    check( pthread_attr_destroy(&pta) );
    return NULL;
}
```

Uitvoer:

print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print2
print2
print2
print2
print2

Source: [pthread.c]

Pthread with Parameter Example (1 of 2)

```
typedef struct {
    char *msg;
    useconds_t us;
} par_t;

void *print(void *par) {
    par_t* p = par;
    for (int i = 0; i < 10; i++) {
        usleep(p->us);
        printf(p->msg);
    }
    return NULL;
}
```

Pthread with Parameter Example (2 of 2)

```
void *main_thread(void *arg) {
    pthread_attr_t pta;
    pthread_attr_init(&pta);
    pthread_attr_setstacksize(&pta, 1024);

    pthread_t t1, t2;
    par_t p1 = {"print1\n", 100000};
    par_t p2 = {"print2\n", 200000};
    pthread_create(&t1, &pta, &print, &p1);
    pthread_create(&t2, &pta, &print, &p2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    check( pthread_attr_destroy(&pta) );
    return NULL;
}
```

Uitvoer:

```
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print2
print2
print2
print2
print2
```

Source: [pthread_par.c]()

Problem with Shared Memory

Source: pthread_shared.c

```
volatile int aantal = 0;

void *teller(void *par) {
    for (int i = 0; i < 10000000; i++) {
        aantal++;
    }
    return NULL;
}

//…
    pthread_create(&t1, &pta, &teller, NULL);
    pthread_create(&t2, &pta, &teller, NULL);
    pthread_create(&t3, &pta, &teller, NULL);
```

What is the final value of aantal?

Problem with Shared Memory

- The operation `aantal++` is **not atomic** (in machine code).
  - For example, `X10` contains the address of `aantal`:

```
LDUR X9, [X10, #0]
ADDI X9, X9, #1
STUR X9, [X10, #0]
```

What happens when a task switch occurs at this moment?

- What is the minimal and the maximal final value of aantal?
  - Minimum = 10000000
  - Maximum = 30000000

Data Corruption

Situations: Task A and B use a shared global variable
(just demonstrated)

Task C and D are both using the same peripheral
(e.g., UART port)

Goal: Preventing concurrent use of a resource by multiple tasks

Solution: Using tokens to represent resources. Allow a limited number
of tasks to get the same token at the same time.

exceed expectations

HOGESCHOOL ROTTERDAM

Solution?

- There are solutions which use shared variables (2 flags and 1 turn variable) and **busy waiting**.
  - Dekker's algorithm: http://en.wikipedia.org/wiki/Dekker's_algorithm
  - Peterson's algorithm: http://en.wikipedia.org/wiki/Peterson's_algorithm
- Busy waiting **costs** clock cycles!
- OSes offer solutions **without** busy waiting.

IPC Inter Process (Task) Communication

- Shared variable based
  - Busy waiting
    - Inefficient
    - Mutual exclusion is hard (Dekker's or Peterson's algorithm)
  - Spinlock
    - Busy waiting
  - Mutex
  - Semaphore
  - Monitor
    - Mutex combined with Conditional variables
  - Barrier
  - Read Write Lock
  - Event Groups

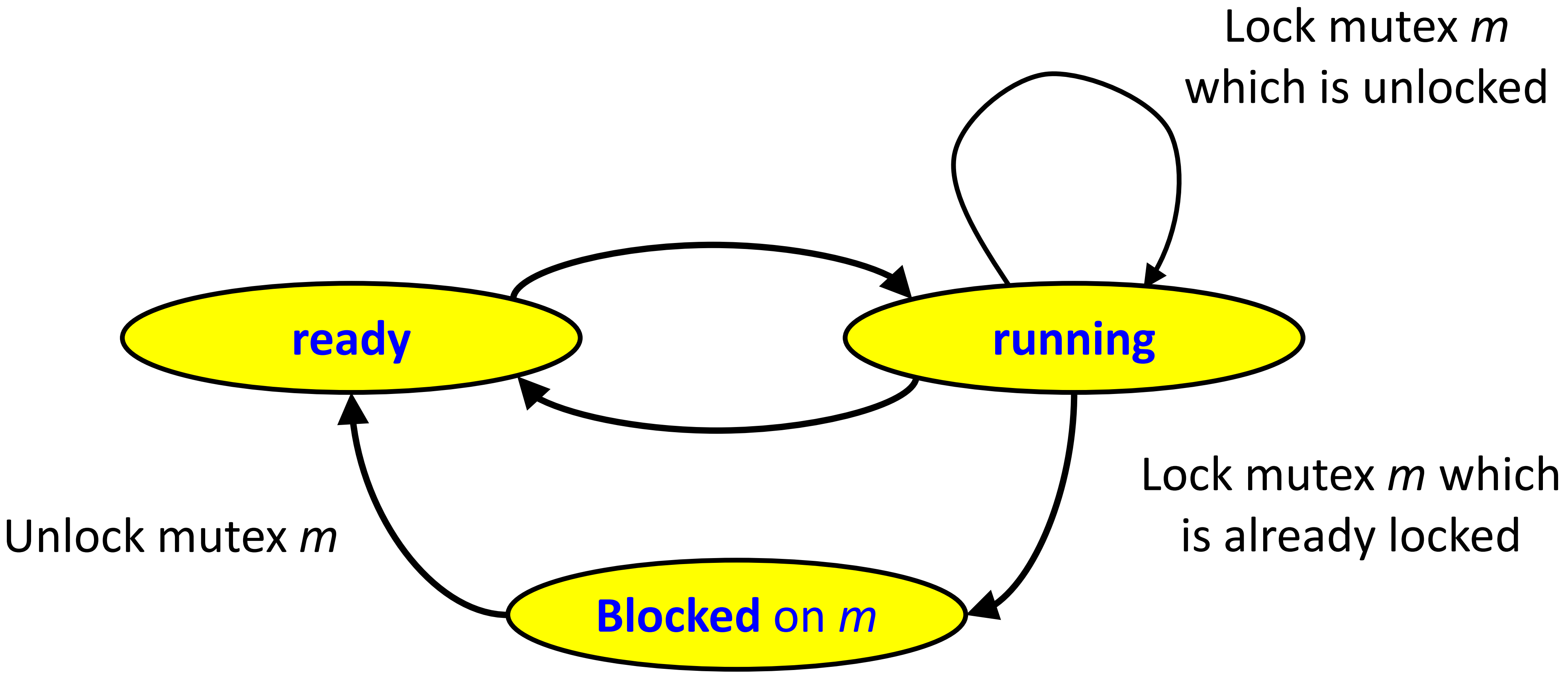
- Message based
  - Message Queue

Mutex

- Simple way to create a **mutual exclusive** so-called critical section.
  - Only **one** task can be in the critical section.

- Mutex has a **lock** (take) and a **unlock** (give) function.
  - OS ensures that these functions are **atomic**!
  - At the start of the critical section the mutex must be locked (taken) and at the end of the critical section the mutex must be unlocked (given).

Task States



exceed expectations

HOGESCHOOL ROTTERDAM

Mutex

- When a task *t* tries to lock mutex *m* which is already locked by an other task, task *t* is blocked on *m*.
  We also say:
  - Task *t* waits for mutex *m*.
  - Task *t* sleeps until mutex *m* is unlocked.

- Order of unblocking (waking up):
  - general purpose OS: FIFO
  - real-time OS: highest priority

Mutex with Shared Memory

```c
int aantal = 0;
pthread_mutex_t m;

void *teller(void *par) {
    for (int i = 0; i < 10000000; i++) {
        pthread_mutex_lock(&m);
        aantal++;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}
```

Source: [mutex.c]()

Data Corruption

DANGER

- Priority inversion
  - Low priority task has mutex locked
  - High priority task is blocked due to mutex
  - Solution: priority inheritance

Will be discussed in week 7!

- Deadlock
  - Task A has resource 1 locked and wants to lock resource 2
  - Task B has resource 2 locked and wants to lock resource 1

Counting Semaphore

- Operations:
  - Psem (prolaag (probeer te verlagen), take, wait):
    wait (block, sleep) if `count == 0` else decrement `count`.
  - Vsem (verhoog, signal, give, post):
    unblock a waiting task if `count == 0` else increment count.

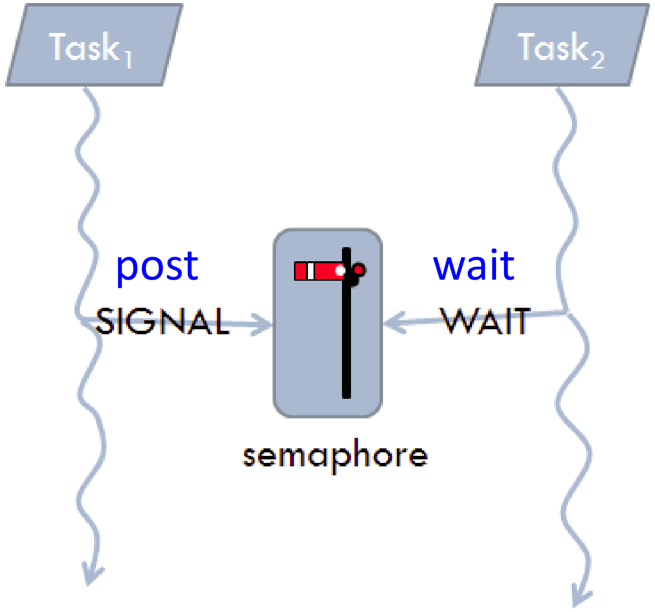
- Order of unblocking (wake up):
  - general purpose: FIFO
  - real-time: highest priority

Edsger Dijkstra

Semaphore versus Mutex

- Mutex can only be used for mutual exclusion (task which takes the mutex should also give the mutex (back)).
- Semaphore can also be used for other synchronization purposes.

- Homework:
  - Task *a* consists of two sequential parts a_1 and a_2.
  - Task *b* consists of two sequential parts b_1 and b_2.
  - Task *c* consists of two sequential parts c_1 and c_2.
  - Make sure (using a semaphore) that the parts b_2 and c_2 are always executed after part a_1 has been executed.



exceed expectations
HOGESCHOOL ROTTERDAM

Inter Task Communication

Situation: Task A reads/debounces buttons

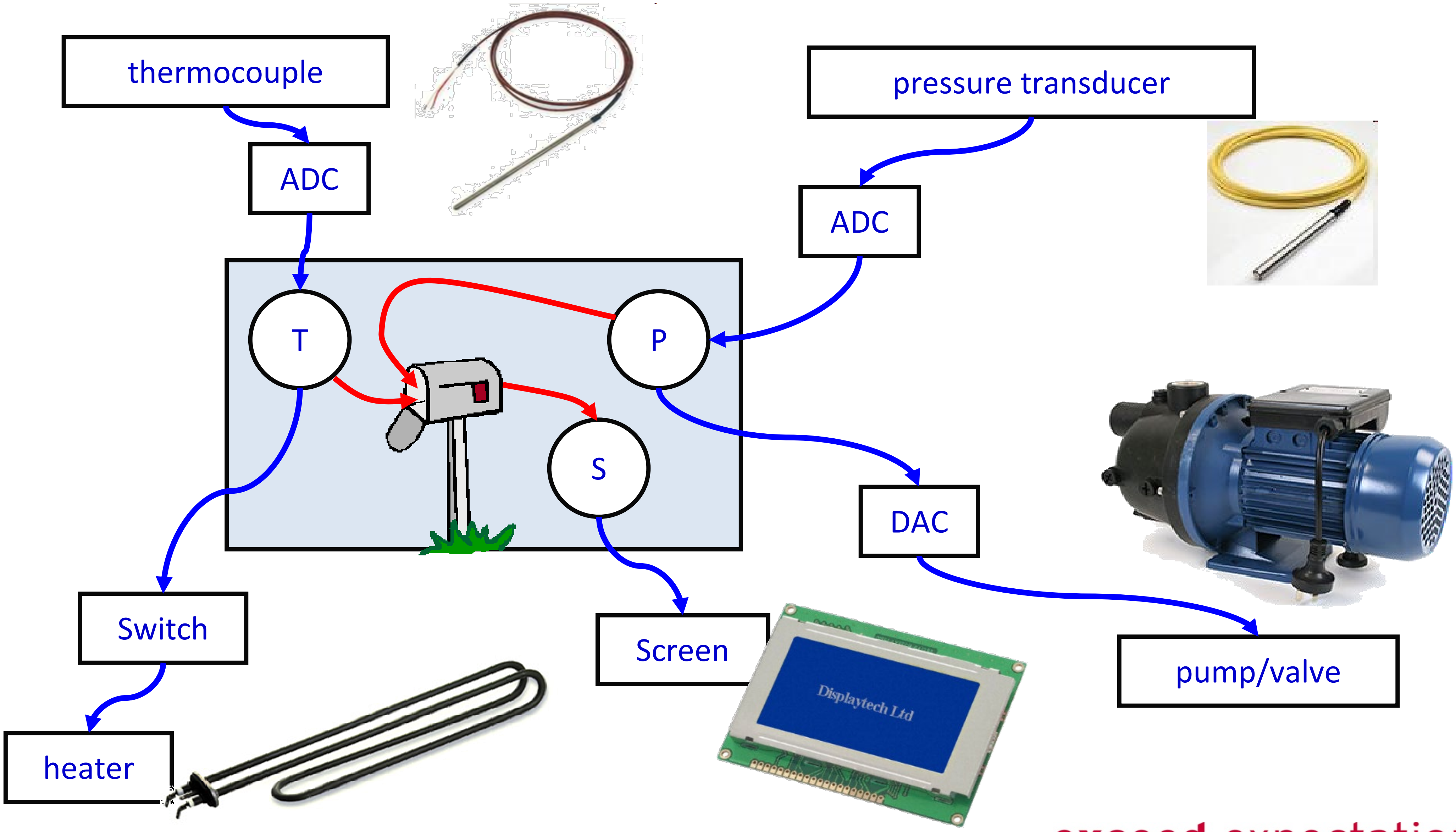
Task B executes functionality based on button pressed

Task C is the gate to the USB port

Other tasks send messages to C

Goal: Create a message queue variable that tasks can add to and receive from.

Communication between Threads



Message Queue Example (1 of 2)

```
void *main_thread(void *arg)
{
    mqd_t mqdes;
    mq_attr mqAttrs;
    mqAttrs.mq_maxmsg = 3;
    mqAttrs.mq_msgsize = sizeof(int);
    mqAttrs.mq_flags = 0;
    mqdes = mq_open("ints", O_RDWR | O_CREAT, 0666, &mqAttrs);

    pthread_t tp, tc;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setstacksize(&attr, 1024);
    pthread_create(&tp, &attr, &producer, &mqdes);
    pthread_create(&tc, &attr, &consumer, &mqdes);
```

Message Queue Example (2 of 2)

```
void *producer(void *p) {
    mqd_t mq = *(mqd_t *)p;
    for (int i = 0; i < 10; i++) {
        mq_send(mq, (char *)&i, sizeof(i), 0);
    }
    return NULL;
}
void *consumer(void *p) {
    mqd_t mq = *(mqd_t *)p;
    for (int i = 0; i < 10; i++) {
        int msg;
        mq_receive(mq, (char *)&msg, sizeof(msg), NULL);
        printf("%d\n", msg);
    }
    return NULL;
}
```

Source: [mqueue.c]

Next Week

- Week 6: Schedulability Analyses, Priority Assignment
- Week 7: Response Time Analyses