Real-Time Embedded Systems

DT8025, Fall 2016

http://goo.gl/AZfc9l

Lecture 3

Masoumeh Taromirad

m.taromirad@hh.se

HÖGSKOLAN I HALMSTAD

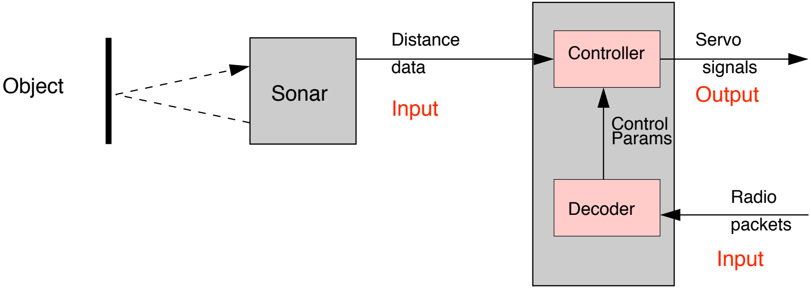
Center for Research on Embedded Systems

School of Information Technology

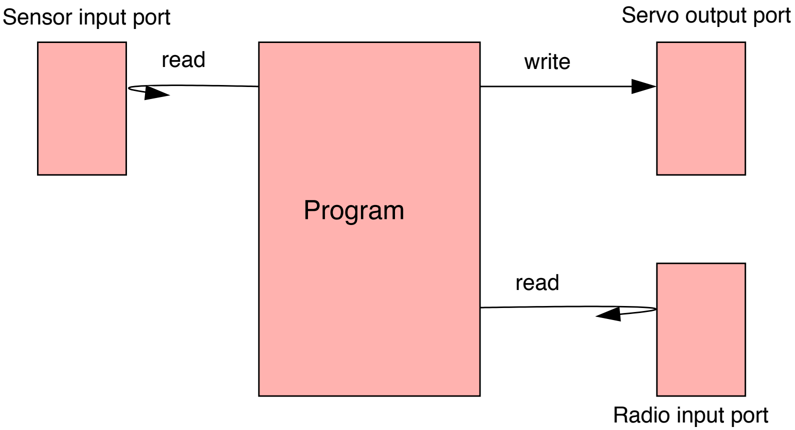
A simple embedded system revisited!

Need for Concurrency

Follow an object using sonar echoes.
Control parameters sent over wireless.
The servo controls wheels.



The view from the processor



The program: a first attempt

```
main(){
   struct Params params;
   struct Packet packet;
   int dist, signal;
   while(1){
     dist = sonar_read();
     control(dist, &signal, &params);
     servo_write(signal);

     radio_read(&packet);
     decode(&packet,&params);
   }
}
```

The program: input

```
int sonar_read(){
    while(SONAR_STATUS & READY == 0);
    return SONAR_DATA;
}
```

Functions creating an *illusion* to the rest of the program!

```
void radio_read(struct Packet *pkt){
  while(RADIO_STATUS & READY == 0);
  pkt->v1 = RADIO_DATA1;
  ...
  pkt->vn = RADIO_DATAn;
}
```

Assuming that status is automatically reset when data is read.

The program: operations & output

Control

Calculates the servo signal.

```
void control(int dist, int *sig, struct Params *p);
```

Decode

Decodes a packet and calculates new control parameters

```
void decode(struct Packet *pkt, struct Params *p)
```

Output

Writes to the servo controls

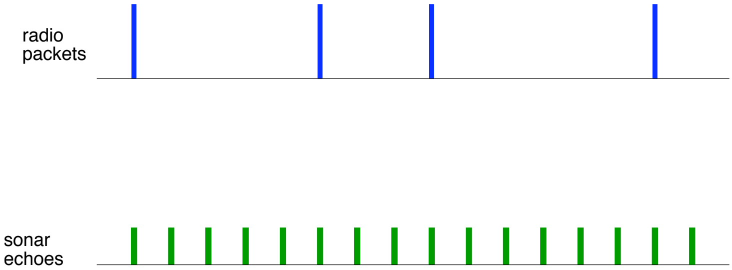
```
void servo_write(int sig){
  SERVO_DATA = sig;
}
```

The program: busy waiting input

```
int sonar_read(){
   while(SONAR_STATUS & READY == 0);
   return SONAR_DATA;
}
```

```
void radio_read(struct Packet *pkt){
  while(RADIO_STATUS & READY == 0);
  pkt->v1 = RADIO_DATA1;
  ...
  pkt->vn = RADIO_DATAn;
}
```

Problems?



Problem: Unknown and unrelated frequencies of events

Ignoring the other event while busy waiting!

Why busy waiting

- Data is not already in place (. . . radio packets are not!)
- Even if there might be reasons for waiting, sensors may provide no (useful) content!
- They *produce* data only because they are asked to (. . . remote transmitters act autonomously!)

- RAM and files vs. external input
- Memory-mapped I/O may give the wrong *illusion*!

The program: a second attempt

```
while(1){
 if(SONAR_STATUS & READY){
   dist = SONAR_DATA;
  control(dist,&signal,&params);
  servo_write(signal);
 }
 if(RADIO_STATUS & READY){
   packet->v1 = RADIO_DATA1;
    ...;
   packet->vn = RADIO_DATAn;
  decode(&packet,&params);
 }
}
```

Destroy the functions for reading and have *only one* busy waiting loop!

Centralized busy waiting

Breaking modularity:

- Checking both events in one big busy-waiting loop
- Complicating the simple read operations

100% CPU usage, no matter how frequent input data arrives.

Try to make the main loop run less often!

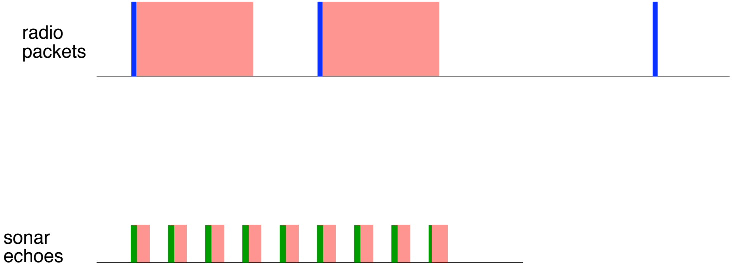
The program: a third attempt

The cyclic executive

```
while(1){
  sleep_until_next_timer_interrupt();
  if(SONAR_STATUS & READY){
     dist = SONAR_DATA;
    control(dist,&signal,&params);
    servo_write(signal);
  }
  if(RADIO_STATUS & READY){
     packet->v1 = RADIO_DATA1;
     ...;
     packet->vn = RADIO_DATAn;
    decode(&packet,&params);
  }
}
```

Compromise: power consumption vs. response time

Problems?



Issue: different duration (processing time) of tasks

Concurrency

What we need

Different parts of a program conceptually execute simultaneously.

Why concurrent execution?

- improve responsiveness
- improve performance
- directly control the timing of external interactions

Concurrency

Why...

- Improve responsiveness
  by avoiding situations where long-running programs can block a program that responds to external stimuli (e.g. sensor data or a user request).
  Improved responsiveness reduces **latency**.
- Improve performance
- Directly control the timing of external interactions.
  at that time.

Concurrency

Why...

- Improve responsiveness
- Improve performance
  by allowing a program to run simultaneously on multiple processors or cores.
- Directly control the timing of external interactions.
  at that time.

Concurrency

Why...

- Improve responsiveness
- Improve performance
- Directly control the timing of external interactions.
  A program may need to perform some action, such as updating a display, at particular times, regardless of what other tasks might be executing at that time.

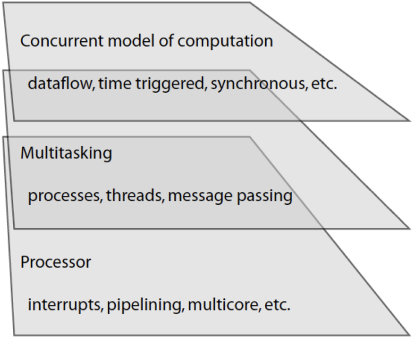
Concurrency addresses timing issues.

Concurrency

Layers of Abstraction

Multitasking

- mid-level techniques
- implemented using the low-level mechanisms
- supporting concurrent execution of multiple tasks.



Concurrency

concurrent execution of sequential code

Possible solution: task interleaving

Seizing control and allowing for other tasks to take over: interleaving task fragments.

Challenges

- concurrent execution of sequential code
- a solution for different frequencies (and the waiting time)

Interleaving by hand

```
void decode(struct Packet *pkt, struct Params p){
   phase1(pkt,p);
   try_sonar_task();
   phase2(pkt,p);
   try_sonar_task();
   phase3(pkt,p);
}
```

```
void try_sonar_task(){
  if(SONAR_STATUS & READY){
   dist = SONAR_DATA;
   control(dist,&signal,&params);
   servo_write(signal);
  }
}
```

Again, breaking modularity in an ad-hoc way. How many phases of decode are sufficient?

Interleaving by hand

More fine breaking up might be needed . . .

```
void phase2(struct Packet *pkt, struct Params *p){
   while(expr){
     try_sonar_task();
     phase21(pkt,p);
   }
}
```

Interleaving by hand

More fine breaking up might be needed ...

```
void phase2(struct Packet *pkt, struct Params *p){
   int i = 0;
   while(expr){
     if(i%800==0)try_sonar_task();
     i++;
     phase21(pkt,p);
   }
}
```

Unstructured and ad-hoc; any better alternative?

About Practical 1

In lab 1 you will program 3 functions

- Test-Driven Development of an algorithm to calculate the exponential function e^x,
- porting the function to write on the display (PiFace Display),
- interleaving the blinker with the function, and
- modify the interleaving to keep the blinking period intact.

Automatic interleaving?

low-level concurrency

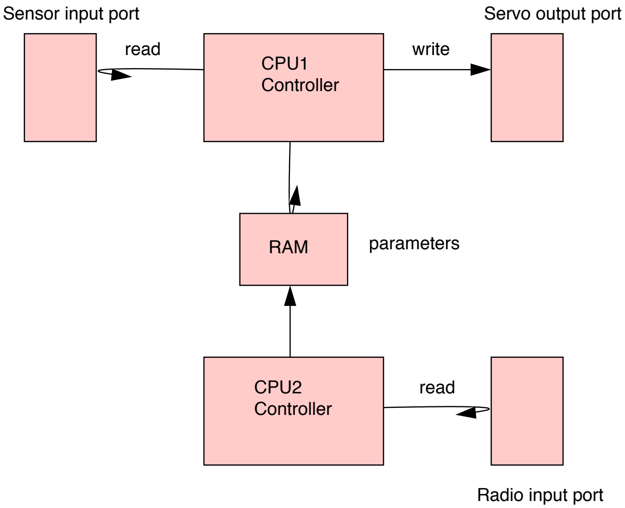
There are 2 tasks, driven by independent input sources.

Handle sonar echoes running the control algorithm and updating the servo.

Handle radio packets by running the decoder.

Had we had access to 2 CPUs we could place one task in each. We can imagine some construct that allows us to express this in our program.

Two CPUs


Sensor input port
read
CPU1
Controller
write
Servo output port
RAM
parameters
CPU2
Controller
read
Radio input port

Two CPU's program

```
struct Params params;
```

```
void controller_main(){
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main(){
   struct Packet packet;
   while(1){
      radio_read(&packet);
      decode(&packet,&params);
   }
}
```

We need some way of making one program of this!

Concurrent Programming

Mid-level concurrency

Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems.

A thread is a unique execution of a sequence of machine instructions, that can be interleaved with other threads executing on the same machine.

Threads run concurrently and share a memory space and can access each others' variables.

A system supporting seemingly concurrent execution is called multi-threaded.

Where should threads belong?

A programming language?

As in Java or Ada. Programs are well organized and are independent of the OS.

Libs and OS?

Like C with POSIX threads? Good for multilanguage composition given that OS standards are followed.

Our first multi-threaded program

```
struct Params params;

void controller_main(){
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}

void decoder_main(){
   struct Packet packet;
   while(1){
      radio_read(&packet);
      decode(&packet,&params);
   }
}

main(){
 decoder_main;
 controller_main();
}
```

Our first multi-threaded program

```
struct Params params;

void controller_main(){
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
           &signal,
           &params);
    servo_write(signal);
  }
}

void decoder_main(){
  struct Packet packet;
  while(1){
    radio_read(&packet);
    decode(&packet,&params);
  }
}

main(){
  cearte_thread(decoder_main);
  controller_main();
}
```

Mutual Exclusion

It is required that one thread of execution never enters its critical section at the same time that another, concurrent thread of execution enters its own critical section; preventing **race condition** (i.e., two concurrent pieces of code race to access the same resource).

Threads

Main issues and challenges

Mutual Exclusion

Scheduling

The core of an implementation of threads is a scheduler that decides which thread to execute next when a processor is available to execute a thread.

Threads

Main issues and challenges

Mutual Exclusion

Scheduling

Context Switch

The process of storing and restoring the state (more specifically, the execution context) of a process or thread so that execution can be resumed from the same point at a later time.

Threads

Main issues and challenges

Mutual Exclusion

It is required that one thread of execution never enters its critical section at the same time that another, concurrent thread of execution enters its own critical section; preventing **race condition**.

Scheduling

Context Switch

Our first multi-threaded program

```
struct Params params;

void controller_main(){
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}

void decoder_main(){
   struct Packet packet;
   while(1){
      radio_read(&packet);
      decode(&packet,&params);
   }
}

main(){
  cearte_thread(decoder_main);
  controller_main();
}
```

The critical section problem

What will happen if the `params` struct is read (by the controller) at the same time it is written (by the decoder)?

I.e., what if the scheduler happens to insert some decoder instructions while some, but not all, of the controller's reads have been done?

This problem is central to concurrent programming where there is any ammount of sharing!

Critical sections in real life

Car dealer

Displays used car

Puts up price tag

Displays luxury car

Updates price tag

Car buyer

Becomes interested,sells her old car

Gets angry!

Critical sections in real life

Car dealer

Displays used car

Puts up price tag

Displays luxury car

Updates price tag

Car buyer

Chooses to keep her old car

All good!

Critical sections in programs

Imagine uppdating the same bank account from two places at approximately the same time (e.g. your employer deposits your salary at more or less the same time as you are making a small deposit).

```
                    int account = 0;
account = account + 500;    account = account + 10000;
```

When this is compiled there might be several instructions for each update!

Critical sections in programs

```
load account,r1
add 500,r1
store r1, account
```

```
load account, r2
add 10000, r2
store r2, account
```

Final balance is 10500

Critical sections in programs

```
load account, r2
add 10000, r2
store r2, account
```

```
load account,r1
add 500,r1
store r1, account
```

Final balance is 10500

Critical sections in programs

```
load account,r1
                              load account, r2
                              add 10000, r2
add 500,r1
                              store r2, account
store r1, account
```

Final balance is 500

Critical sections in programs

Testing and setting

```
int shopper;
```

```
if(shopper == NONE)                 if(shopper==NONE)
  shopper = HUSBAND                     shopper = WIFE
```

Possible interleaving

```
if(shopper == NONE)
                                    if(shopper==NONE)

shopper = HUSBAND
                                    shopper = WIFE
```

Our embedded system

Exchanging parameters

```
              struct Params p;
while(1){                    while(1){
  ...                          local_minD = p.minDistance;
  p.minDistance = e1;          local_maxS = p.maxSpeed;
  p.maxSpeed = e2;             ...
}                            }
```

Possible interleaving

```
p.minDistance = 1;
p.maxSpeed = 1;
                                    local_minD = 1;
p.minDistance = 200;
p.maxSpeed = 150;
                                    local_maxS = 150
```

The classical solution

Apply an access protocol to the critical sections that ensures mutual exclusion.

Require that all parties follow the protocol.

Access protocols are realized by means of a shared datastructure known as a mutex or a lock.

The classical solution

A mutual exclusion lock prevents any two threads from simultaneously accessing or modifying a shared resource.

The code between the lock and unlock is a critical section.

At any one time, only one thread can be executing code in such a critical section.

Mutual exclusion

Exchanging parameters

```
              struct Params p;
              mutex m;
while(1){                   while(1){
  ...                         lock(&m)
  lock(&m);                   local_minD = p.minDistance;
  p.minDistance = e1;         local_maxS = p.maxSpeed;
  p.maxSpeed = e2;            unlock(&m)
  unlock(&m);                 ...
                            }
}
```

Bonus Question

Bonus Question

Explain briefly the Peterson's algorithm and describe how it achieves mutual exclusion.

Deadline

Thursday 15/09/2015 at 12:00.

Format

A simple document (e.g. PDF). Don't forget your name!

Email your answers to m.taromirad@hh.se. Beware of plagiarism!

Mutual exclusion

A Challenge

Deadlock

A deadlock occurs when some threads become permanently blocked trying to acquire locks.

Mutual exclusion

A Challenge: Deadlock

```
mutex m1, m2;
```

```
while(1){
  ...
  lock(&m1);
  ...
  lock(&m2);
  ...
  unlock(&m2)
  unlock(&m1);
}
```

```
while(1){
  lock(&m2);
  ...
  lock(&m1);
  ...
  unlock(&m1)
  unlock(&m2);
  ...
}
```

Mutual exclusion

A Challenge: Deadlock

Such deadly embraces have alertno escape. The program needs to be aborted!

Avoid deadlock?

- Deadlock can be difficult to avoid.
- Luckily, there are necessary conditions for deadlock to occur; any of which can be removed to avoid deadlock.

  Example: use only one lock throughout an entire multi-threaded program.

Bonus Question

Bonus Question

Explain briefly (at least three) existing techniques to avoid deadlock in multi-threaded programs.

Deadline

Thursday 15/09/2015 at 12:00.

Format

A simple document (e.g. PDF). Don't forget your name!

Email your answers to m.taromirad@hh.se. Beware of plagiarism!

Threads

Even more problems!

Threads are hard!

- very difficult to understand,
- difficult to build confidence and reason about, and
- yield insidious errors, race conditions, deadlock
  (very important concerns in embedded systems; safety and livelihood of humans)

It is possible but not easy, to construct reliable and correct multi-threaded programs; expert programmers have to be very cautious!