Embedded Systems Programming - PA8001

http://goo.gl/YdEcZU

Lecture 3

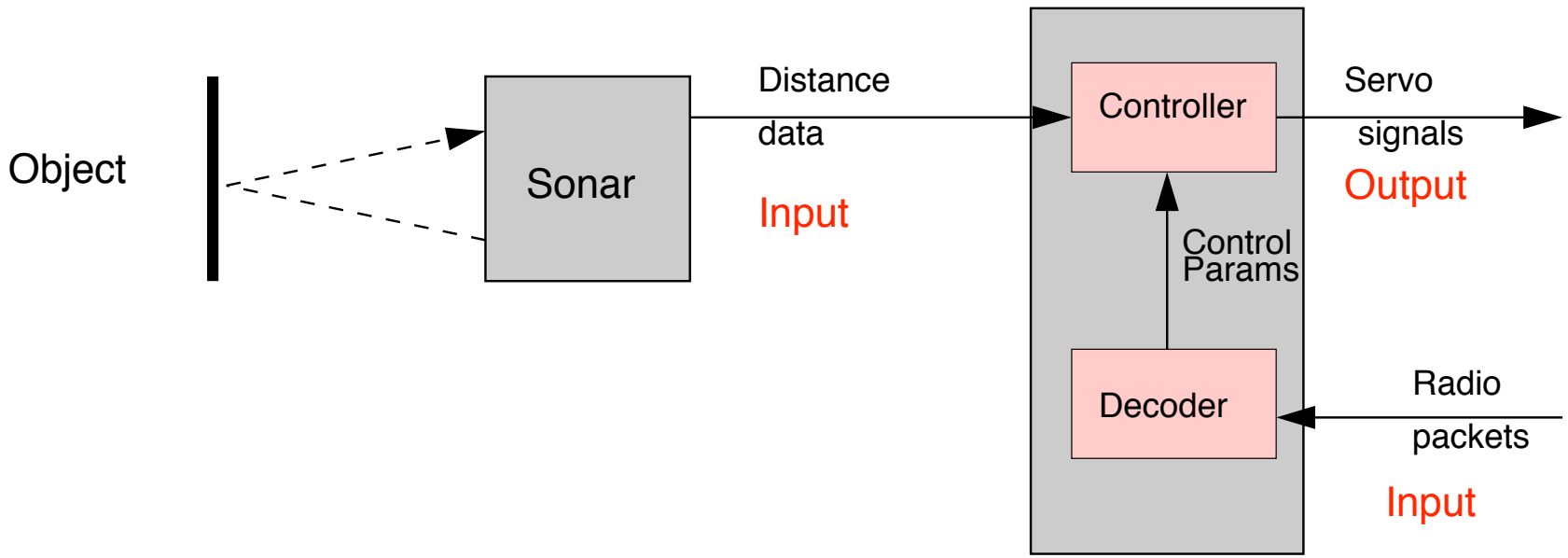
Mohammad Mousavi

m.r.mousavi@hh.se

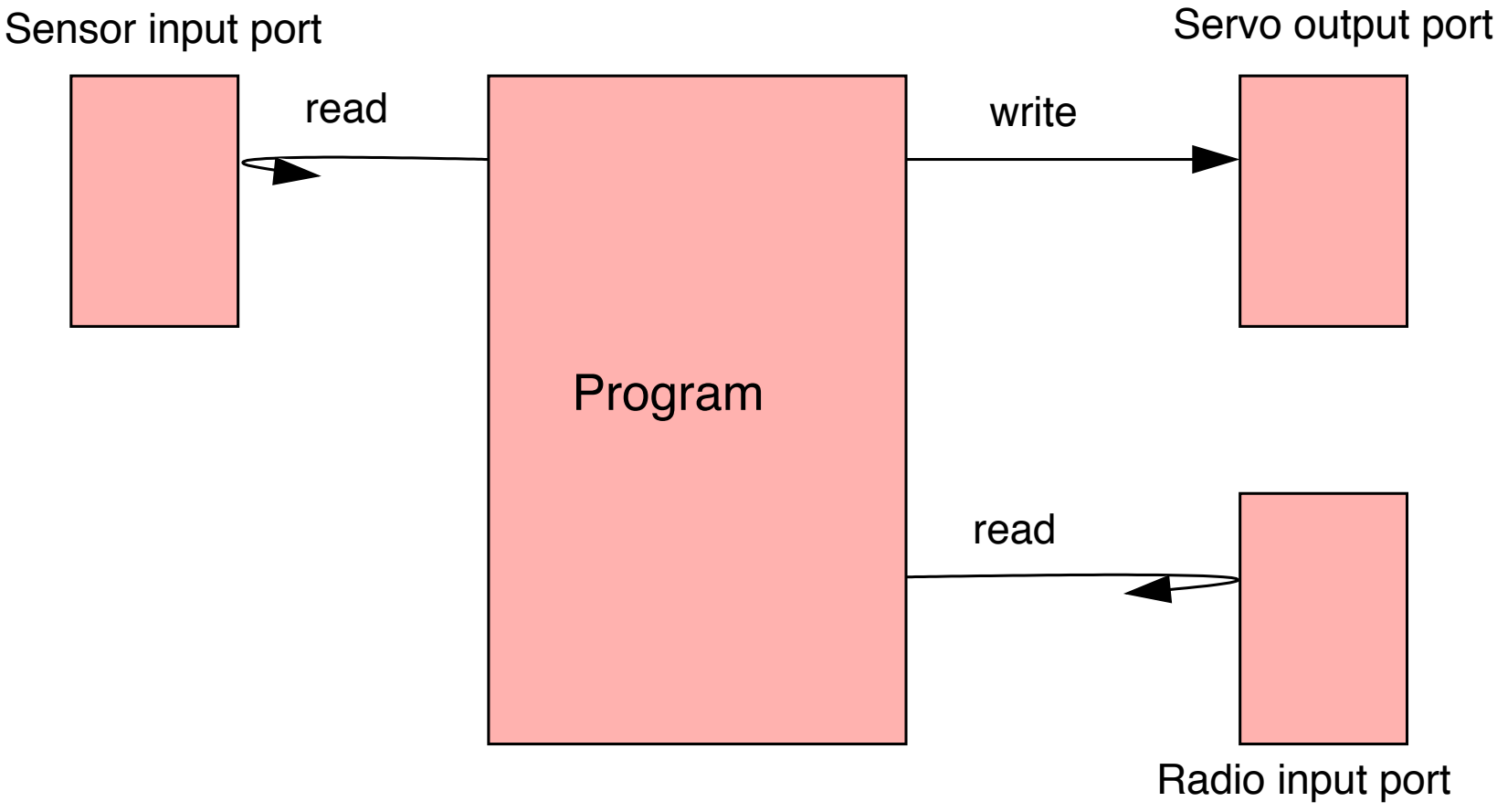
Center for Research on Embedded Systems
School of Information Science, Computer and Electrical Engineering

A simple embedded system

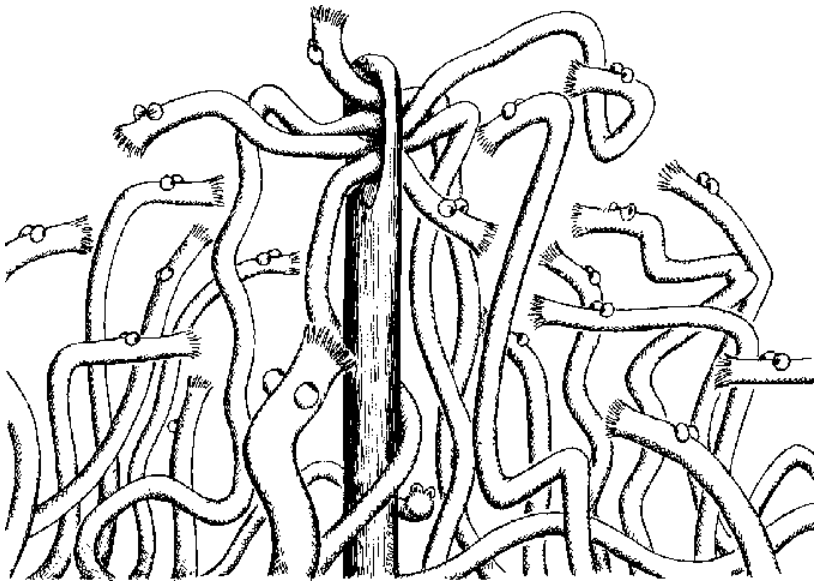
Follow an object using sonar echoes. Control parameters sent over wireless. The servo controls wheels.



The view from the processor



The program



Our order of business:
Concurrency matters!

Even with a single processor, and more so in current parallel architectures.

If time allows...
How to implement threads.

The program: busy waiting 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: output

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

The program: algorithms

Control

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

Calculates the servo signal.

Decode

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

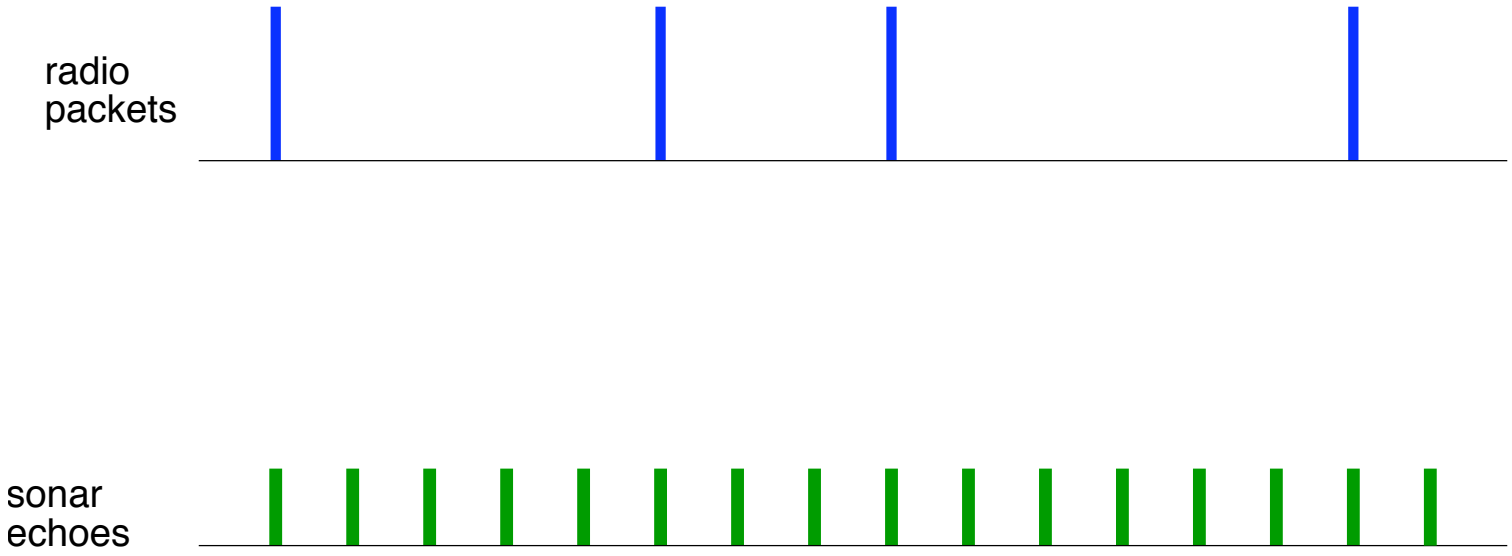
Decodes a packet and calculates new control parameters

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);
    }
}
```

Problems?



Problem: Unknown and unrelated frequencies of events

Ignoring the other event while busy waiting!

The problem explained

RAM and files vs. external input

- Data is 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!)

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!

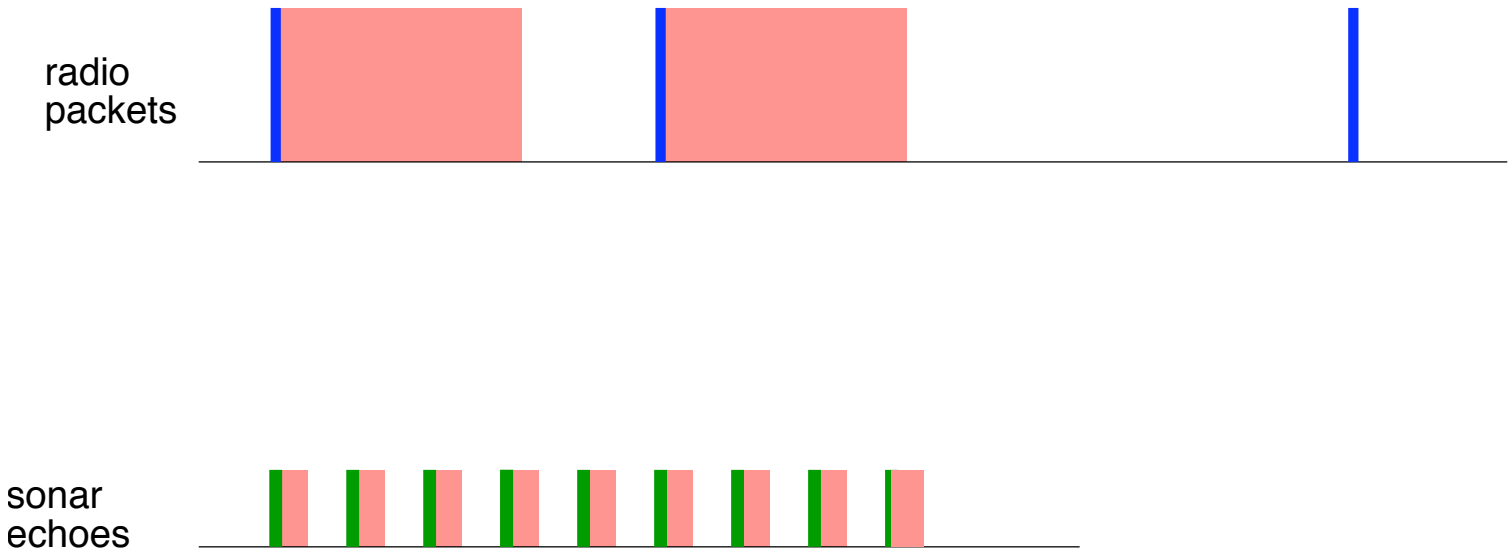
The program: a third attempt

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

Concurrent execution

- Hitherto: a solution for different frequencies (and the waiting time)
- Challenge: concurrent execution

Possible solution

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

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 a square root algorithm,
- Porting square root to write on (the HDMI) display,
- Interleaving the blinker with the square root, and
- Modify the interleaving to keep the blinking period intact.

Automatic interleaving?

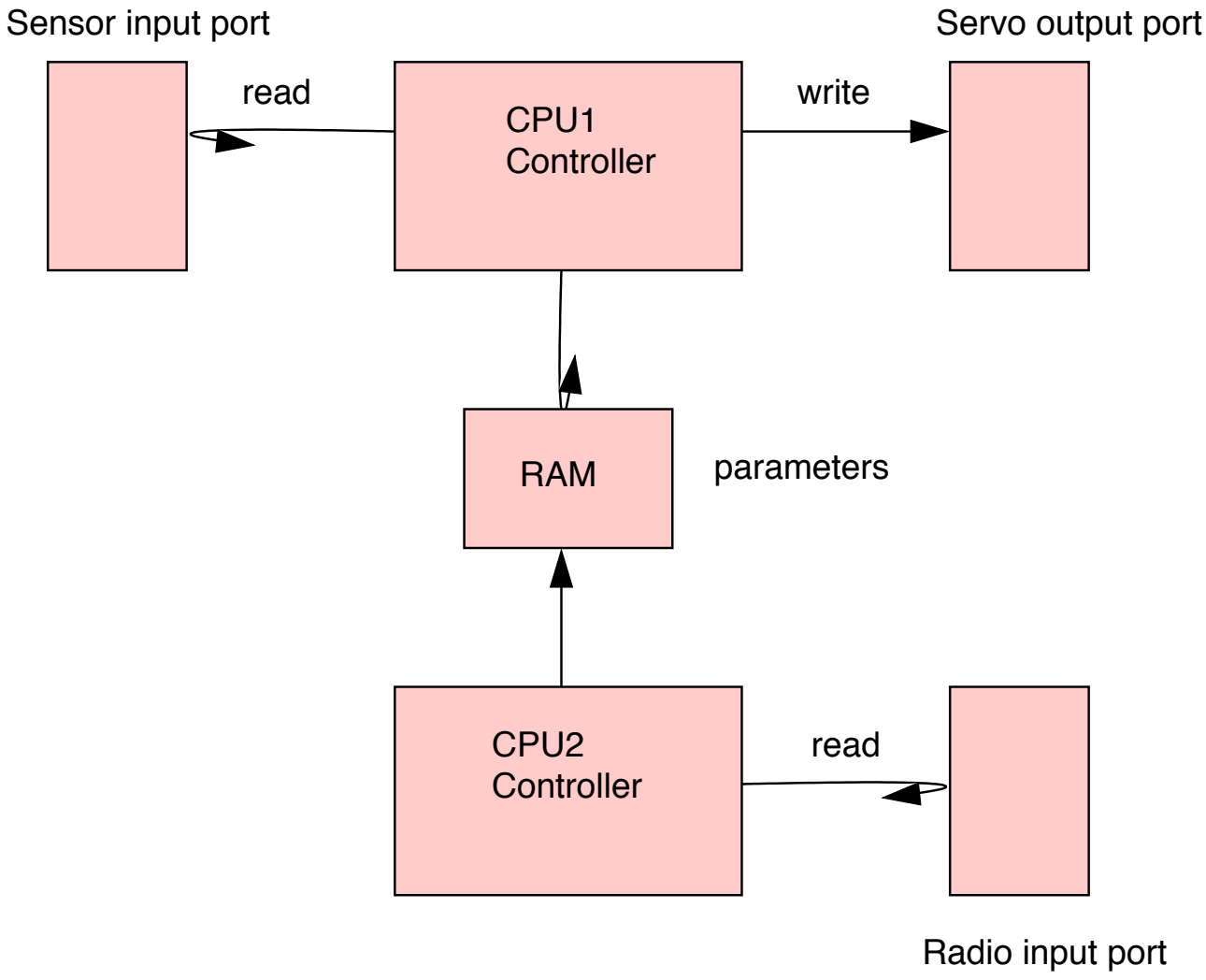
There are 2 tasks, driven by independent input sources.

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
Servo output port
read
write
CPU1
Controller
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! We will deal with it next lecture!

Concurrent Programming

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

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

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

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.

This course - first part

For pedagogical purposes we choose to work with C and a small kernel.

Our first multithreaded 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(){
  spawn(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	**Car buyer**
Displays used car	
Puts up price tag	
Displays luxury car	
	Becomes interested,sells her old car
Updates price tag	
	Gets angry!

Critical sections in real life

Car dealer	**Car buyer**
Displays used car	
Puts up price tag	
Displays luxury car	
Updates price tag	
	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
add 500,r1
store r1, account
```

```
load account, r2
add 10000, r2
store r2, 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 amutex or a lock.

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);                        ...
                                   }
}
```

Classic Example

Bonus Question

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

Deadline

Friday 18/09/2015 at 13:00.