Embedded Systems Programming

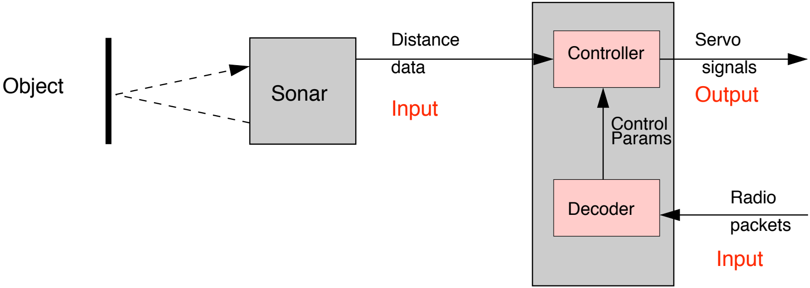
Lecture 3

Verónica Gaspes
www2.hh.se/staff/vero

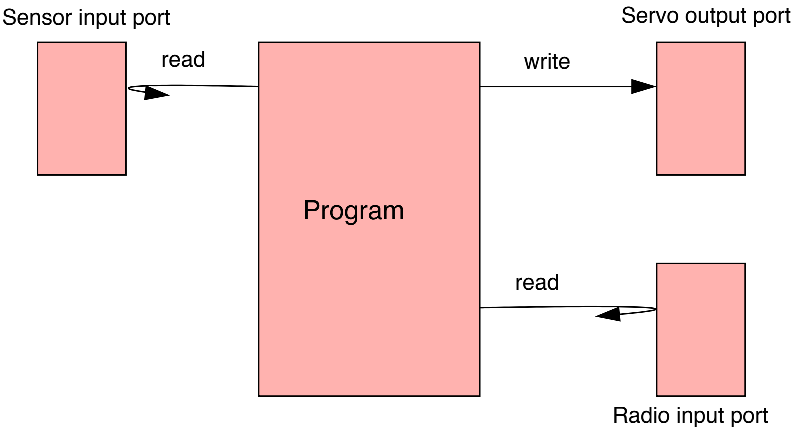
Center for Research on Embedded Systems
School of Information Science, Computer and Electrical Engineering

A simple embedded system

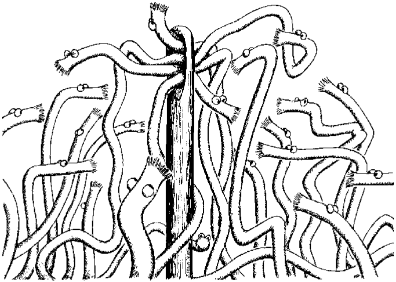
Follow (track) an object using sonar echoes. Control parameters are sent over wireless. The servo controls wheels.



The view from the processor



The program



We will go through a series of attempts to organize the program leading to the need for threads.

We discuss new problems that arise because of programming with threads.

Next lectures

Implementing threads.

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;
}
```

We can define *functions*. that create an *illusion* to the rest of the program!

We have assumed input ports that automatically reset status when data is read.

The program: busy waiting input

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

We can define *functions*. that create 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;
}
```

We have assumed input ports that automatically reset status when data is read.

The program: busy waiting input

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

We can define *functions*. that create 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;
}
```

We have assumed input ports that automatically reset status when data is read.

The program: busy waiting input

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

We can define *functions*. that create 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;
}
```

We have assumed input ports that automatically reset status 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: 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: algorithms

Control

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

Calculates the servo signal.

Decode

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

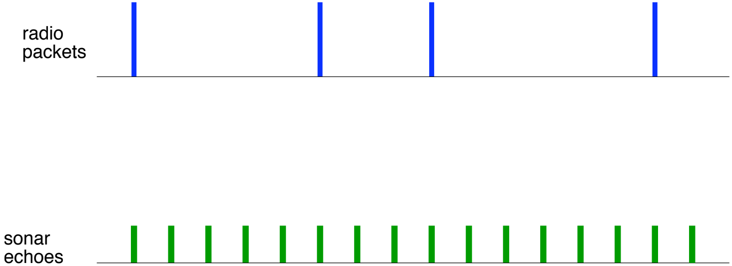
Decodes a packet and calculates new control parameters

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);
   }
}
```

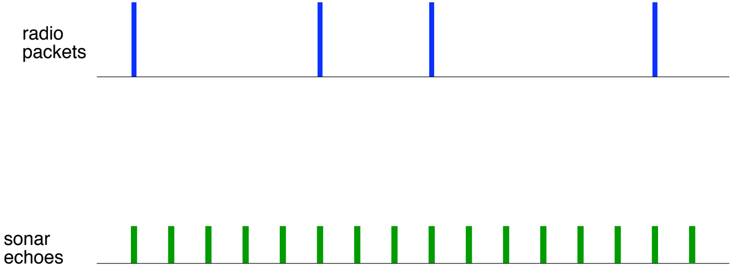
Problems?



We do not know what port will have new data next! The sonar and the radio generate events that are unrelated to each other!

Our program will ignore all events of one kind that happen while busy waiting for the other event!

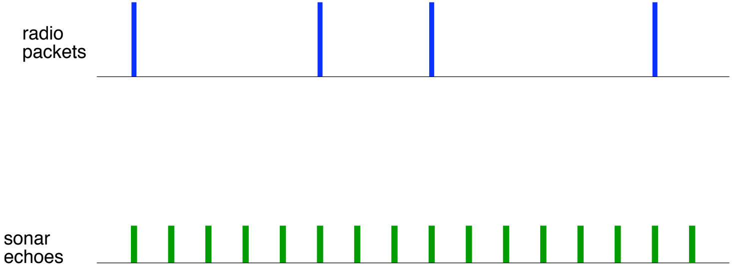
Problems?



We do not know what port will have new data next! The sonar and the radio generate events that are unrelated to each other!

Our program will ignore all events of one kind that happen while busy waiting for the other event!

Problems?



We do not know what port will have new data next! The sonar and the radio generate events that are unrelated to each other!

Our program will ignore all events of one kind that happen while busy waiting for the other event!

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, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

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, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

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, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

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, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

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, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

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->v2 = RADIO_DATAn;
   decode(&packet,&params);
 }
}
```

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

Centralized busy waiting

- The new implementation checks both status registers in one big busy-waiting loop. This avoids waiting for the wrong input.
- We destroyed the simple read operations! VERY not modular!

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

Try to make the main loop run less often!

Centralized busy waiting

- The new implementation checks both status registers in one big busy-waiting loop. This avoids waiting for the wrong input.
- We destroyed the simple read operations! VERY not modular!

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

Try to make the main loop run less often!

Centralized busy waiting

- The new implementation checks both status registers in one big busy-waiting loop. This avoids waiting for the wrong input.
- We destroyed the simple read operations! VERY not modular!

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

Try to make the main loop run less often!

Centralized busy waiting

- The new implementation checks both status registers in one big busy-waiting loop. This avoids waiting for the wrong input.
- We destroyed the simple read operations! VERY not modular!

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

Try to make the main loop run less often!

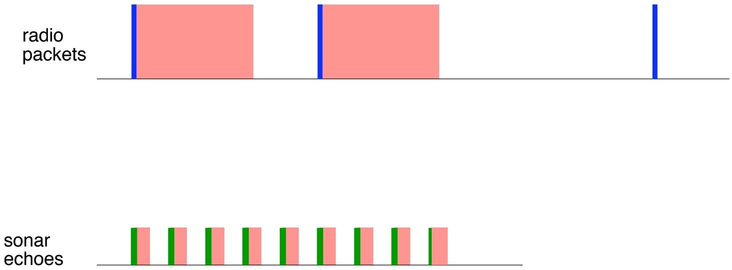
The program: a third attempt

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->v2 = RADIO_DATAn;
    decode(&packet,&params);
  }
}
```

The CPU runns at a fixed rate! The timer period must be set to trade power consumption against task response!

Problems?



If processing time for the infrequent radio packets is much longer than for the frequent sonar echoes . . .

Concurrent execution

- We could solve (in a rather ad-hoc way) how to wait concurrently.
- Now we need to express concurrent execution . . .

¿Imagine . . . . . . that we could interrupt execution of packet decoding when a sonar echo arrives so that the control algorithm can be run. Then decoding could resume! The two tasks fragments are interleaved.

Concurrent execution

- We could solve (in a rather ad-hoc way) how to wait concurrently.
- Now we need to express concurrent execution . . .

–

¿Imagine . . . . . . that we could interrupt execution of packet decoding when a sonar echo arrives so that the control algorithm can be run. Then decoding could resume! The two tasks fragments are interleaved.

Concurrent execution

- We could solve (in a rather ad-hoc way) how to wait concurrently.
- Now we need to express concurrent execution . . .

–

¿Imagine . . . . . . that we could interrupt execution of packet decoding when a sonar echo arrives so that the control algorithm can be run. Then decoding could resume! The two tasks fragments are interleaved.

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 we break tha logical organization of the program in an ad-hoc way! How many phases of decode will we need to run the sonar often enough?

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 we break tha logical organization of the program in an ad-hoc way! How many phases of decode will we need to run the sonar often enough?

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 we break tha logical organization of the program in an ad-hoc way! How many phases of decode will we need to run the sonar often enough?

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 we break tha logical organization of the program in an ad-hoc way! How many phases of decode will we need to run the sonar often enough?

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);
   }
}
```

Code can become very unstructured and complicated very soon.

And then someone might come up with a new, better decoding algorithm ...

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);
   }
}
```

Code can become very unstructured and complicated very soon.

And then someone might come up with a new, better decoding algorithm . . .

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);
   }
}
```

Code can become very unstructured and complicated very soon.

And then someone might come up with a new, better decoding algorithm . . .

About laboration 1

In lab 1 you will program 3 functions

- Writing prime numbers to the LCD
- Blinking with a symbol in the LCD
- Turing on and off a symbol in the LCD using the joystick

In lab 1 you will then write a big program to do all things at once. You will then test the cyclic executive and interleaving by hand (to experience the difficulties with it!)

About laboration 1

In lab 1 you will program 3 functions

- Writing prime numbers to the LCD
- Blinking with a symbol in the LCD
- Turing on and off a symbol in the LCD using the joystick

In lab 1 you will then write a big program to do all things at once. You will then test the cyclic executive and interleaving by hand (to experience the difficulties with it!)

About laboration 1

In lab 1 you will program 3 functions

- Writing prime numbers to the LCD
- Blinking with a symbol in the LCD
- Turing on and off a symbol in the LCD using the joystick

In lab 1 you will then write a big program to do all things at once. You will then test the cyclic executive and interleaving by hand (to experience the difficulties with it!)

About laboration 1

In lab 1 you will program 3 functions

- Writing prime numbers to the LCD
- Blinking with a symbol in the LCD
- Turing on and off a symbol in the LCD using the joystick

In lab 1 you will then write a big program to do all things at once. You will then test the cyclic executive and interleaving by hand (to experience the difficulties with it!)

About laboration 1

In lab 1 you will program 3 functions

- Writing prime numbers to the LCD
- Blinking with a symbol in the LCD
- Turing on and off a symbol in the LCD using the joystick

In lab 1 you will then write a big program to do all things at once. You will then test the cyclic executive and interleaving by hand (to experience the difficulties with it!)

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 contruct that allows us to express this in our program.

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 contruct that allows us to express this in our program.

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 contruct that allows us to express this in our program.

Automatic interleaving?

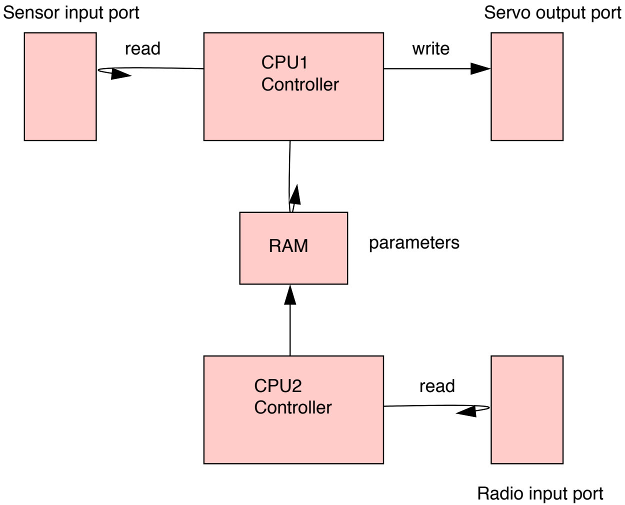
There are 2 tasks, driven by independent input sources.

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 contruct 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! 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.

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.

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.

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.

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.

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!

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!

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

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

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

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

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

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

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

Becomes interested,sells her old car

Gets angry!

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

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 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 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 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 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 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 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

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
```

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
```

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.

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.

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){
  ...
  lock(&m);
  p.minDistance = e1;
  p.maxSpeed = e2;
  unlock(&m);

}
```

```
while(1){
  lock(&m)
  local_minD = p.minDistance;
  local_maxS = p.maxSpeed;
  unlock(&m)
  ...
}
```