Realtime Systems D0003E

Lecture 3: Concurrent threads and mutual exclusion
Burns/Wellings ch. 7 + parts of ch. 8

Overview

• the concurrency problem
• busy-wait
• interleaving code by hand
• thread support
  — automatic interleaving
• critical section problem
  — one solution
  • there are other solutions...

A simple embedded system

Objective: track object by means of sonar echoes, on basis of control parameters received over radio

Initial view

How do we structure the software?

Busy-waiting input & output

Three pieces of code:

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

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

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

• For simplicity
  • assume input ports reset their status bits when data are read
  • rather common case in practice

Pure algorithms

• void control( int dist, int *sig, struct Params *p );
  — Computes a servo signal on basis of its internal state, the given distance, and a set of control parameters

• void decode( struct Packet *pkt, struct Params *p );
  — Decodes a packet to obtain new control parameters
A first attempt

```c
main() {
    struct Params params;
    struct Packet packet;
    int dist, sig;
    while (1) {
        dist = sonar_read();
        control(dist, &sig, &params);
        servo_write(sig);
        radio_read(&packet);
        decode(&packet, &params);
    }
}
```

Input patterns

- sonar echoes and radio packets
  - unrelated events
    - cannot know in advance which will come
  - consequence
    - busy-wait loops in alternating order is clearly ad hoc
      - ignores all the events that might occur in between

Core problem 1

- By coding `sonar_read()` and `radio_read()` using busy-waiting
  - attempt to provide illusion
    - receiving echoes and packets is just a variant of reading data from memory
  - achieved by silently blocking (suspending, halting, freezing) execution
    - when no data is currently present
  - But the RAM memory is an extremely awkward model of an external world that seems to act on its own
  - the RAM model is unable to concurrently wait for multiple events
    - you are forced to choose your source of input (address) before you block

A second attempt

```c
while (1) {
    if (SONAR_STATUS & READY) {
        dist = SONAR_DATA;
        control(dist, &sig, &params);
        servo_write(sig);
    }
    if (RADIO_STATUS & READY) {
        packet.v1 = RADIO_DATA1;
        ...
        packet.vn = RADIO_DATA_n;
        decode(&packet, &params);
    }
}
```

A third attempt

```c
while (1) {
    sleep_until_next_timer_interrupt();
    if (SONAR_STATUS & READY) {
        dist = SONAR_DATA;
        control(dist, &sig, &params);
        servo_write(sig);
    }
    if (RADIO_STATUS & READY) {
        packet.v1 = RADIO_DATA1;
        ...
        packet.vn = RADIO_DATA_n;
        decode(&packet, &params);
    }
}
```

Centralized busy-waiting

- The new implementation
  - checks both status registers in one big busy-waiting loop
  - premature waiting for the “wrong” input is avoided
- The price...
  - breaking up the simple read operations
    - if a third input port must be added one day, the centralized loop must be rewritten
- Moreover, busy-waiting uses 100% CPU
  - irrespective of how frequent input data arrive
    - (think power consumption in cell-phones!)
- Perhaps if the main code could be run less often...

A second attempt

- what if we break up the encapsulated read operations
  - merge the busy-wait loops?
- All problems solved?

A third attempt

- Let’s assume we have an inline assembler primitive that literally puts the CPU to sleep until the next timer interrupt (we’ll study the details later in the course)
  - while (1) {
    sleep_until_next_timer_interrupt();
    if (SONAR_STATUS & READY) {
        dist = SONAR_DATA;
        control(dist, &sig, &params);
        servo_write(sig);
    }
    if (RADIO_STATUS & READY) {
        packet.v1 = RADIO_DATA1;
        ...
        packet.vn = RADIO_DATA_n;
        decode(&packet, &params);
    }
  }
- The furious behavior during resting periods is now controlled.
- Any remaining problems?
The cyclic executive

- The shown program pattern is called the cyclic executive
  - very common in small embedded systems
- Basic idea
  - run a set of non-blocking tasks in sequence in a big loop
  - executed at a fixed rate
- counter variables can be used to limit certain tasks to every N turns of the loop only
- The timer period
  - trade power consumption against desired task response times
  - in the degenerate case we're back to busy waiting
- However, now let’s take the execution time of tasks into consideration...

Differing time scales

- radio packets
- sonar echoes
  - radio packets
    - infrequent
    - require long processing
  - sonar echoes
    - frequent
    - require less processing
  - Now what will happen to sonar echo processing once the program has started decoding a radio packet?

Core problem 2

- core problem 1
  - inability to express concurrent waiting
- now
  - inability to express concurrent code execution
- real concurrency
  - cannot be achieved on a single CPU
  - anyway...
- one could wish...
  - temporarily interrupt the packet decoding function
    - when a sonar echo arrives, in order to run the control algorithm
  - coding up such interleaved functionality by hand might seem a little daunting! Let’s see why!

Interleaving by hand

- If we’re lucky, decode has a simple structure:

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

- If we're less lucky, phase1, phase2, & phase3 must be split up as well, so that try_sonar_task will be run often enough

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

Interleaving by hand

- If we’re even less lucky, algorithm loops must be broken too:

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

Interleaving by hand

```
void phase3(struct Packet *pkt, struct Params *p) {
  while (expr) {
    phase31(pkt,p);
  }
```

Interleaving by hand

```
```
Interleaving by hand

- check sonar every 800th iteration:

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

- In general, a trade-off might be needed between checking for input every turn, and not checking inside the loop at all.

Interleaving by hand

- it doesn’t end here:
  - conditional branches (if-statements)
  - execution time may vary substantially

```c
void phase2937(struct Packet *pkt, struct Params *p) {
    int flagX = 0, flagY = 1, flagZ = 0, count = 10000;
    phase2937_a(pkt, p, &flagY, &count);
    if (count%1800 == 0 || flagX && !flagZ) try_sonar_task();
    ...
    phase2937_g(pkt,p, &flagY);
    if (flagY==flagX && pkt->bytes[333] != 0)
    try_sonar_task();
}
```

Interleaving by hand

- Then we realize
  - in some situations, the control algorithm must be broken up too

```c
void try_radio_task() {
    if (radio_ready()) {
        radio_read(&pkt);
        decode(&pkt, &p);
    }
}
```

```c
void decode(...) {
    phase1(pkt,p);
    try_sonar_task();
    phase2(pkt,p);
    try_sonar_task();
    phase3(pkt,p);
}
```

```c
void control(...) {
    ... if (...) try_radio_task();
    ...
}
```

Interleaving by hand

- And then the radio communication expert wants to try out a redesigned algorithm for packet decoding...

```c
Back to basics

- two separate tasks, independent input sources:
  - Handle sonar echoes
  - control algorithm and updating the servo
  - Handle radio packets
    - decoder

- if we could express each of these tasks separately...
  - imagine double cpu’s
  - write one program for each cpu

A double CPU system

```

- two separate tasks, independent input sources:
  - Handle sonar echoes
  - control algorithm and updating the servo
  - Handle radio packets
    - decoder

- if we could express each of these tasks separately...
  - imagine double cpu’s
  - write one program for each cpu

A double CPU system

```

- two separate tasks, independent input sources:
  - Handle sonar echoes
  - control algorithm and updating the servo
  - Handle radio packets
    - decoder

- if we could express each of these tasks separately...
  - imagine double cpu’s
  - write one program for each cpu

A double CPU system

```

- two separate tasks, independent input sources:
  - Handle sonar echoes
  - control algorithm and updating the servo
  - Handle radio packets
    - decoder

- if we could express each of these tasks separately...
  - imagine double cpu’s
  - write one program for each cpu

A double CPU system

```
A double CPU solution

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

This software is arguably much clearer!

Simulating double CPUs

• Now, if we can’t afford a double CPU system – maybe our single CPU could be built so as to
  • run one instruction from one program
  • then switch to the other one for one instruction
  • then switch back...
  • Or maybe it could run somewhat bigger chunks between switches, to reduce any switching penalties...
  • In fact, this would be like having the previous interleaved program constructed automatically, by some behind-the-scenes scheduler!
  • What would it take...

The C programming model

stack

Locals of a
x = 9
w = 6
v = 0
u = 0
globals

code

a() { int x;
  x = 9;
  w = 6;
  b(55);
}

Then an abrupt switch to...

Locals of b
y = 55
globals

Locals of g
w = [1,1,1,3]
i = 1
v = 0
w = 6
u = 0
globals

Then switching back to...

Locals of g
w = [1,1,1,3]
i = 1
v = 0
w = 6
u = 0
globals

The C programming model

The C programming model

The C programming model

The C programming model

Then switching back to...

Then an abrupt switch to...

And so on...
Threads
- System supporting such seemingly concurrent execution
  - Called multi-threaded
- A thread
  - Unique execution of a sequence of machine instructions
  - Can be interleaved with other threads executing on the same machine
- Each thread
  - Its own execution stack, where its local variables, function arguments, and return addresses are stored
- Shared between threads
  - Global variables, so called static variables
  - Heap-allocated data
  - All other system resources

Automatic interleaving
- Should threads be supported by
  - The programming language?
    - Examples: Ada, Java
    - Pros: readability, OS independence
  - Libraries and the operating system?
    - Examples: C, C++ with POSIX threads
    - Pros: OS standards, multi-language composition
- We choose C and our own kernel for illustrative purposes
  - But the debate continues... :-)
- The technical details will be deferred to next lecture
  - Assume there is an operation spawn...

A threaded solution
```c
struct Params params;

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

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

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

Well, we still depend on busy-waiting, but we'll improve on that later

The critical section problem
- What will happen
  - Params struct is read (by the controller)
  - At the same time it is written (by the decoder)?
- I.e., what if scheduler inserts some decoder instructions
  - Some, but not all, of the controller's reads have been done?
- Old and new parameters
  - Risk of being mixed...
- This is the critical section problem
  - Fundamental to concurrent programming

Critical sections in real life

<table>
<thead>
<tr>
<th>Car dealer</th>
<th>Car buyer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displays used car</td>
<td>Displays luxury car</td>
</tr>
<tr>
<td>Puts up price tag</td>
<td>Updates price tag</td>
</tr>
<tr>
<td>Displays luxury car</td>
<td>Becomes interested, sells his old car</td>
</tr>
<tr>
<td>Updates price tag</td>
<td>Gets angry :-(</td>
</tr>
</tbody>
</table>

Critical sections in real life

<table>
<thead>
<tr>
<th>Car dealer</th>
<th>Car buyer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displays used car</td>
<td>Displays luxury car</td>
</tr>
<tr>
<td>Puts up price tag</td>
<td>Updates price tag</td>
</tr>
<tr>
<td>Displays luxury car</td>
<td>Chooses to keep his old car</td>
</tr>
<tr>
<td></td>
<td>No reason to get angry</td>
</tr>
</tbody>
</table>
Critical sections in programs

Updating an account

```c
int account = 0;
account = account + 3;  account = account + 4;
```

Possible interleaving:

```c
load account,r1
add 3,r1
store r1,account
```

```c
load account,r2
add 4,r2
store r2,account
```

End balance is 7

The classic solution

- Apply an “access protocol” to the critical sections — ensures mutual exclusion
- Require that all parties follow the protocol
- Essentially a way of asking the thread scheduler not to perform certain context switches
- Concretely, the access protocols are realized by means of a shared data structure known as a mutex (or a lock)
Mutual exclusion

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

Impossible interleaving:
```
p.minDistance = 200;
p.maxSpeed = 150;
```

Cannot run, because lock is already taken!

Wrapping up so far

- The spawn primitive
  - run several tasks in a pseudo-concurrent fashion
- The lock and unlock primitives make it possible to solve the critical section problem
- We still depend on busy-waiting for input
  - with multiple timers we could probably achieve the effect of multiple cyclic executives
- With these tools we’ve actually introduced the essentials of classical shared memory concurrent programming

Shared memory prog.

- Basic design method:
  - create a thread wherever blocking behavior or execution times call for concurrency
  - then apply a mutex protocol wherever a critical section is revealed
- Perhaps not surprisingly, this programming model can be rather brittle, as it doesn’t suggest much structure
- In particular, the critical sections can in general be hard to spot
- Central question: how do we modularize? According to thread or critical section structure?

Our next path

- Instead of learning all about programming with spawn, lock and unlock, we’ll investigate how they are realized inside a kernel
- Then we’ll quickly jump to a more abstract view of concurrent systems, where
  - objects and messages are the primary notions instead of threads and locks
  - spawn, lock and unlock are only used inside the kernel
  - critical sections are automatically protected
  - busy-waiting can be avoided altogether
- Then, at the end of the course, we’ll revisit traditional concurrent programming in the light of our new OO model