NSWI120 - Page **1/2** (exam #5 – 2017.02.13)

Write your answers to the special response sheet you received (with your name and photograph). If you are using more than a single sheet of paper for your answers, then mark each sheet with its number / total number of sheets you will hand over.

Task 1

Assume a typical modern operating system line Linux or Windows. Describe and explain what would be the main parts of a process context.

Task 2

Assume we are implementing an I^2C ambient light sensor that should communicate via the same protocol as sensor ALS-PDIC17-57B/TR8 of the Everlight company is using (its datasheet is attached to this exam sheet – focus mainly on page 11). When designing our sensor we came across a problem that occurs if master is communicating with our device using a `100 kHz` clock frequency. If we are accepting a 16-bit word read transaction, then after successfully receiving the address byte and after its positive acknowledgement we are never able to respond sooner than after 0.1 milliseconds with the following transmission of the first ADC register byte. Explain in detail, if (and how) it is possible to solve this problem using the features provided by the I^2C bus.

Task 3

Assume the following declarations (`longword` is a 32-bit unsigned integer type, `word` is a 16-bit unsigned integer type):

```
  type PLongword = ^longword;
       PWord = ^word;
  procedure Conv(src : PWord; dst : PLongword);
```

Implement the `Conv` procedure in Free Pascal, so that it translates the input null-terminated string `src` from UTF-16 LE encoding into UTF-32 LE encoding and it stores the resulting UTF-32 LE null-terminated string characters to a memory location pointed to by the `dst` argument (assume your code will be run only on little-endian platforms, and that the target location for the `dst` variable has enough unused space). Assume Unicode characters in range `$00D800` to `$00DFFF` can never appear in any valid Unicode string, and can be used by a concrete encoding to encode other characters. Also assume that codes above `$10FFFF` have no valid character assigned in Unicode, and that Unicode characters in range `$010000` to `$10FFFF` are in UTF-16 encoded as follows:

(1) First subtract value `$010000` from character's code, and then the resulting 20-bit number is split into two 10-bit parts that are separately encoded according the following rules.

(2) The upper 10 bits of the 20-bit value are stored in the 10 lowest bits of the first 16-bit surrogate character (stored on lower address). Upper 6 bits of the first surrogate are set to (bit 15 is leftmost, bit 10 is rightmost):

```
1101 10
```

(3) The lower 10 bits of the 20-bit value are stored in the lowest bits of the second 16-bit surrogate character (stored on higher address). Upper 6 bits of the second surrogate are set to (only bit 10 is different from first surrogate):

```
1101 11
```

Task 4

We are founders of Matfyz' first fast food chain called *MFC* (*Malostranský Fried Cheese*). The main refreshment is a wide variety of beverages produced by *Pepsi Co*. We plan to use the same model for selling the beverages as our closest competitor (*KFC*) is using, i.e. our customers will be able to tank any amount of a selected beverage into a provided cup. Our goal is to manufacture a post mix device for our restaurants, that should use a proven design – it will use a

touch screen (see picture) with `576x1024` pixels resolution, that should always display a list of available beverages (each represented by a square `180x180` pixel area on the screen) – any liquid should pour from the post mix if and only if there is a continuous touch detected over a beverage image. The output nozzle has several parallel inputs: pure tap water (controlled by an electromagnetic valve `Vwater`), carbonated water (controlled by valve `Vsoda`), and 5 nozzles from tanks with specific flavors (valves `V1` to `V5`). The following beverages are available (braces contain coordinates of the top left corner of the controlling square and number of the control valve): *Pepsi* (`198,60→1`), *Pepsi Light* (`85,280→2`), *7UP* (`310,280→3`), *Mirinda* (`85,490→4`), *Lipton Ice Tea* (`310,490→5`) – only ice tea is mixed with pure water, everything else is mixed with soda. There are also options to tank pure water (`85,700`) or pure soda (`310,700`).

We will be using a 32-bit μC to control the post mix – the controller has an embedded I^2C bus controller and also 32 bit GPIO controller with digital I/O lines `IO0` to `IO31`. We have connected `IO0` to valve `Vwater`, pins `IO1` to `IO5` → valves `V1` to `V5`, pin `IO6` → valve `Vsoda`. All GPIO lines are initially configured as output lines inside the controller, and we have a predefined procedure that can be used to manipulate their state (LSbit represents state of `IO0`, MSbit represents state of `IO31`):

```
procedure SetGpioOutputs(pins : longword);
```

There is a touch screen layer controller connected to the I^2C bus. We have available the following function:

```
function GetTouchInfo(
  var x : longword; var y : longword) : boolean;
```

that is using the I^2C bus controller to read the last event detected by the touch screen (x and y are coordinated of the event, return value: `true` = touch was detected, `false` = loss of touch was detected). Implement in Pascal a firmware for the specified computer so that is behaves exactly according the specification of the post mix from above – write the program as simple as possible just by polling the GPIO and touch screen controllers.

Common for all tasks marked with an X

Assume we have a specification of a CLR (Common Language Runtime = standard .NET VM) based virtual machine. Our VM is a machine with **Harvard** and **stack-based register** architecture (all registers contain **32-bit signed** integer numbers; size of the register stack is unlimited). The machine code for this VM is called CIL code (Common Intermediate Language). Assume both the CIL code and the virtual machine is storing all data in **little-endian** order, and that VM's instruction set contains at least the following instructions (all have a single byte opcode followed by argument values [if any]):

- `ldsfld` (opcode `0x7E`) – load static field = load from a global variable, whose address is the only instruction's argument
- `stsfld` (opcode `0x80`) – store static field = store to a global variable, whose address is the only instruction's argument
- `call` (opcode `0x28`) – procedure or function call (arguments are passed left to right on top of the register stack, and are **removed by the callee**; return value is also passed on top of the register stack)
- `ret` (opcode `0x2A`) – return from subroutine (no ex. args.)
- `add` (opcode `0x58`) – addition (no explicit arguments)
- `mul` (opcode `0x5A`) – multiplication (no explicit arguments)
- `ble` (opcode `0x31`) – relative conditional jump: branch if less than or equal to: instruction removes two values from the stack, and if the second removed value is lower or equal to the first removed one, a jump to a target location (passed as instruction argument) is made. Opcode is followed by a 1 byte representing jump offset relative to the first byte of the instruction following `ble`.

Task 5 (X)

Without using any inline assembler write in Pascal code of a procedure that, if compiled by a typical Pascal compiler, could result in the following CIL code that we have disassembled from a CIL executable into CIL assembler:

```
  0x20580000 7E 00 00 60 20  ldsfld [0x20600000]
  0x20580005 7E 04 00 60 20  ldsfld [0x20600004]
  0x2058000a 58              add
  0x2058000b 7E 08 00 60 20  ldsfld [0x20600008]
  0x20580010 7E 0C 00 60 20  ldsfld [0x2060000C]
  0x20580015 7E 10 00 60 20  ldsfld [0x20600010]
  0x2058001a 5A              mul
  0x2058001b 58              add
  0x2058001c 31 26           ble 0x20580044
  0x2058001e 7E 08 00 60 20  ldsfld [0x20600008]
  0x20580023 7E 00 00 60 20  ldsfld [0x20600000]
  0x20580028 7E 04 00 60 20  ldsfld [0x20600004]
  0x2058002d 58              add
  0x2058002e 7E 00 00 60 20  ldsfld [0x20600000]
  0x20580033 7E 04 00 60 20  ldsfld [0x20600004]
  0x20580038 5A              mul
  0x20580039 28 00 10 58 20  call 0x20581000
  0x2058003e 58              add
  0x2058003f 80 00 00 60 20  stsfld [0x20600000]
  0x20580044 2A              ret
```

Task 6 (X)

Implement in Pascal a simplified core of the described VM as a CIL code interpreter for the following 3 instructions (rest will be implemented in future): `ldsfld`, `stsfld`, `add`. Assume you have already a working implementation of a stack data structure implemented as a single linked list of `longint` values – see code below. Empty stack is represented by a `nil` value in its top. The `Push` procedure appends new value on top of the stack pointed to by the `top` argument that is then updated by the procedure to point to the new top. The `Pop` function returns current value on the stack top and updates it to point to next item in stack or to `nil`.

```
  type PReg = ^TReg;
    TReg = record
      value : longint;
      next : PReg;
    end;
  procedure Push(
    var top : PReg; value : longint); forward;
  function Pop(var top : PReg) : longint; forward;
```

Assume the CIL code for VM execution in already stored in your global variable `cil`, that represents the code address space of the VM (assume the first instruction for execution lays on address [index] `0`). There is also a global variable `data` representing the data address space.

```
var cil : array[0..$40000000] of byte;
    data : array[0..$40000000] of byte;
```

Task 7

Assume we would like to store real number `12.6875` into a single Pascal variable in our program in the standard *fixed-point* 8.24 representation. How do we define such a variable in standard Pascal? We will execute program containing this variable on a 32-bit little-endian CPU, and store the value `12.6875` into it, and we found out that the variable is then stored on address `0x02BC4300`. Write value of every byte occupied by the variable as a number in hexadecimal system.

Task 8

Assume we are programming a Pascal application whose source code is split into several files (`a1.pas` to `a3.pas`, where `a2.pas` contains the main program body). Explain in detail how and in which steps the final executable file is generated when using a typical Pascal compiler. Which instruction will be pointed to by the *entrypoint* of such an executable file?

Task 9

Assume a common operating system like Linux or Windows running on the x86 architecture (IA-32, i.e. the 32-bit variant of the standard Intel architecture used in common PCs). Describe all the important states an application thread can be in. For each such state explain its meaning and also explain how can a thread typically get into it (e.g. via which kernel syscall).

Task 10

Describe and explain all the necessary signals that a typical parallel memory bus has to have, so that it can be used to connect a single `2048x16` SRAM memory module.