Practicum 4

De scheduler – deel 1

In dit practicum zullen we de scheduler van ons rudimentair besturingssysteem aanpassen zodat meerdere taken kunnen worden uitgevoerd op de computer. In practicum 3 was het slechts mogelijk 1 enkele taak uit te voeren, namelijk *spiraal*. Vanzelfsprekend kunnen deze taken in een uniprocessorsysteem niet tegelijk worden uitgevoerd, maar door het snel wisselen tussen taken, krijgt de gebruiker toch de indruk dat de taken simultaan lopen.

4.1 Indienen

Deze voorbereiding wordt gequoteerd. Je moet zowel je code als een verslag indienen voor (**15 april 2013 om 22:00:00**) via `https://indianio.ugent.be`. Let op! We verwachten per groep één enkel zip bestand waarin de volgende bestanden zijn opgenomen.

- `practicum4_deel1_verslag.pdf`

- `practicum4_deel1_vraag6.asm`

De verbetering zal gebeuren voor de aanvang van het 4e practicum zodat jullie de feedback kunnen gebruiken bij het oplossen daarvan.

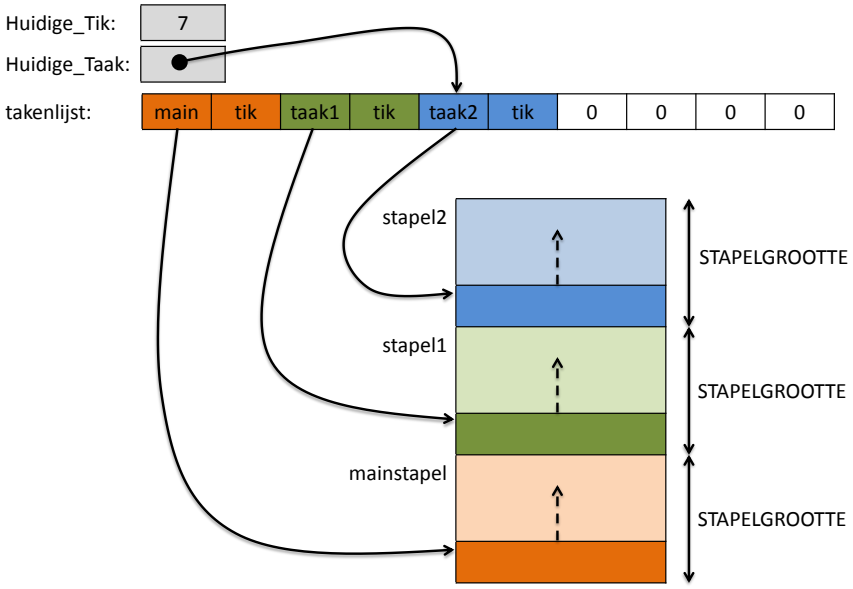
Zoals steeds gelden de algemene richtlijnen voor het indienen van verslagen.

4.2 Inleiding

We vertrekken van de Bochs-omgeving zoals deze gebruikt werd in practicum 3. De bedoeling van dit practicum is om het programma te wijzigen zodat er in plaats van één spiraal, simultaan 2 spiralen getekend worden. We zullen dit niet doen door de routine spiraal aan te passen zodat ze de gevraagde twee spiralen tegelijk tekent, maar wel door de routine spiraal tweemaal op te starten (met verschillende parameters), en heel snel tussen die twee routines te wisselen.

De onderstaande stukjes code kunnen gebruikt worden om twee spiralen op te starten (een linkerspiraal en een rechterspiraal). Elk van deze twee taken zal over een eigen stapel moeten beschikken (zie verder).

```
Taak1: ; tekent een rechterspiraal
  mov eax,40
  mov ebx,79
  mov ecx,0
  mov edx,20
  jmp spiraal
```



Figuur 4.1: Voorstelling van de takenlijst en de stapels die ermee geassocieerd worden.

```
Taak2: ; tekent een linkerspiraal
  mov eax,0
  mov ebx,39
  mov ecx,0
  mov edx,20
  jmp spiraal
```

Het afwisselen tussen de twee spiralen gebeurt als volgt. Op geregelde tijdstippen genereert de hardware van de computer een *timeronderbreking*, die door het besturingssysteem wordt onderschept. Telkens wanneer er een dergelijke onderbreking optreedt, zullen we alle registers van de taak van de (eerste) spiraal die op dat ogenblik getekend wordt, bewaren op de stapel van die taak en de finale versie van de stapelwijzer bewaren in de takenlijst. Dan veranderen we de stapelwijzer zodat hij wijst naar de stapel van de taak behorende bij de andere (tweede) spiraal. Op die stapel werden eerder de registers van die taak bewaard, zodat we nu alle registers kunnen herstellen, waarop de andere (tweede) taak uitvoert en de andere (tweede) spiraal verder getekend wordt. Als er dan opnieuw een timeronderbreking optreedt, vindt een analoge operatie plaats, waarbij nu terug wordt gegaan naar de eerste taak.

We lichten kort de datastructuren toe die door de scheduler worden gebruikt. In wezen beschikt de scheduler over een lijst van taken, waarin de structuur wordt gebruikt die ook kort in de slides van de les i.v.m. bordoefeningen assembler werd toegelicht, zie ook Figuur 3.1:

```
struct taak_entry {
  void* stapelwijzer;
  int tik;
}

struct taak_entry takenlijst[MAX_TAKEN];
```

Deze lijst zal steeds op een circulaire manier worden doorlopen. Bij alle taken, behalve de huidige taak, zal de toestand van de processor (de registers) op de stapel staan. Bij het aanmaken van een nieuwe taak moeten we er dus voor zorgen dat de initiële stapel de correcte beginwaarden van de registers bevat (de meeste hiervan zijn 0, behalve het vlaggenregister, de stapelwijzer en het codesegment). Daarnaast wordt voor elke taak bijgehouden op welke kloktik ze mag geactiveerd worden.

De onderbrekingsroutine die de taakwisseling uitvoert ziet eruit als volgt:

```
schedulerhandler:
        pushad
        inc     dword [Huidige_Tick]
        mov     al, 0x20
        out     0x20, al
        sti
        mov     ebx, [Huidige_Taak]
        mov     dword [ebx],esp
        mov     dword [ebx + 4], 0
        mov     ecx, [Huidige_Tick]
        cli
        mov     esp, 0
.taakzoeklus:
        add     ebx, 8
        cmp     ebx, takenlijst + (MAX_TAKEN * 8)
        jl      .nog_niet_aan_het_einde
        lea     ebx, [takenlijst]
.nog_niet_aan_het_einde:
        cmp     dword [ebx],0
        je      .taakzoeklus
        cmp     dword [ebx+4], ecx
        jg      .taakzoeklus
        mov     [Huidige_Taak], ebx
        mov     esp, [ebx]
        popad
        iret
```

De eerste taak van deze routine is het bewaren van de registers voor algemeen gebruik op de stapel van de onderbroken taak (pushad). Vervolgens wordt de waarde op het adres Huidige_Tick verhoogd met 1 en laat de onderbrekingsregelaar weten dat de onderbreking ontvangen werd (**mov** + **out**). Daarna wordt de top van de stapel van de onderbroken taak in de takenlijst geschreven op de plaats aangewezen door Huidige_Taak. We zorgen ervoor dat de taakzoeklus niet onderbroken kan worden door gebruik van een cli instructie in de lus. Indien je zelf over de takenlijst itereert moet je zelf ook steeds zeker zijn dat de onderbrekingen uit staan.

Vervolgens zoekt de scheduler naar een volgende taak die in aanmerking komt om uitgevoerd te worden. Zodra het einde van de lijst wordt bereikt, wordt de zoektocht verdergezet aan het begin van de lijst. We gebruiken de datastructuur dus als een circulaire lijst. Taken met een kloktik groter dan Huidige_Tick komen nog niet in aanmerking om uitgevoerd te worden en ook in dit geval wordt er verder gezocht.

Eenmaal de scheduler een geschikte taak gevonden heeft, wordt de stapelwijzer van de taak hersteld, worden de registers hersteld (**popad**) en wordt de taak verdergezet op de plaats waar ze oorspronkelijk onderbroken werd (met **iret**). In dit schema veronderstellen we wel aan dat er minstens 1 taak zal gevonden worden. Zoniet zal de scheduler blijven zoeken.

Het enige verschil met een gewone onderbrekingsroutine is dus dat in dit geval niet de onderbroken taak, maar een andere taak verdergezet wordt.

Om te zien dat de schedulerhandler wel degelijk wordt opgeroepen, kun je bijvoorbeeld de vol-

gende code toegevoen in de handler, na de sti instructie:

```
push 0
push 0
push dword [Huidige_Tick]
call printhex
add esp, 12
```

4.3 Opgaven

1. De instructies **pushad** en **popad** worden vaak gebruikt. Wat doen deze instructies? Teken de stapel na de uitvoering van een **pushad** instructie (je mag ervan uitgaan de de stapel initieel leeg is).

2. Vervang de **pushad** instructie door equivalente code (zonder gebruik te maken van **pushad** zelf). Schrijf deze code hieronder neer.

3. Bestudeer het schedulingmechanisme en annoteer deze code. We verwachten zowel een duidelijke uitleg over het functionele aspect van de instructies (wat doet de instructie) als een beschrijving van de semantiek (waarom staat die instructie daar, wat is de bedoeling, hoe past deze instructie in het volledige plaatje? Als voorbeeld kan je de geannoteerde code van `lucas` gebruiken. Deze vraag is vooral een hulp bij het oplossen van de volgende oefening, en zal niet gequoteerd zijn.

4. Beschrijf op een hoog abstractieniveau in pseudo-code of in C, Java, ... het algoritme van de scheduler. Gebruik hierbij dus abstracte concepten zoals array-indexering, duidelijke functienamen voor low-levelfunctionaliteit zoals **returnFromInterrupt**, **saveAllRegisters**, ..., en duidelijke variabelenamen voor registers, indien mogelijk (dus als je bijvoorbeeld weet dat register **esi** een index in de takenlijst bevat, gebruik dan bijvoorbeeld **TaakIndex**, ...

5. Verklaar waarom er in de schedulerhandler een **cli** instructie zonder corresponderende **sti** instructie voorkomt. Is dit een fout? **Indien wel**: waar dient de **sti** precies geplaatst te worden in de code? **Indien niet**: leg uit waarom niet.

6. Installeer de schedulerhandler op de timeronderbreking, en zet de timeronderbrekingen aan, zodat het programma nu een aantal keer per seconde zal onderbroken worden en na het aflopen van de lijst opnieuw zal opgestart worden. Zorg ervoor dat het programma blijft lopen zoals voorheen. In principe mag je de talrijke onderbrekingen niet opmerken.

   Sla jouw code op als `practicum4_deel1_vraag6.asm`.

Succes!