Student: **Collegekaartnummer:**

Tentamen Computersystemen voor AI programmeurs

baiCSAI3 2e jaar bachelor AI, 2e semester 25 mei 2011

vraag 1

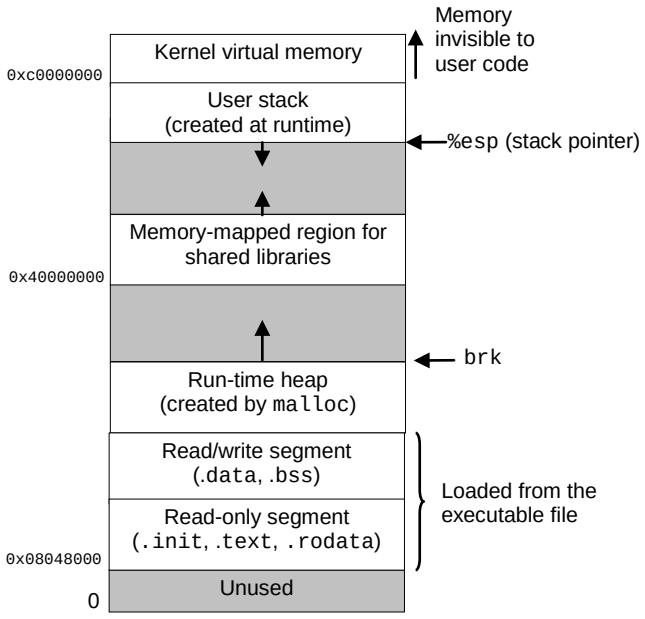
Met de machine-instructie CPUID kun je bij IA32-processoren informatie opvragen over de processor. Het type van informatie dat in de vier registers `eax, ebx, ecx, edx` wordt weggeschreven kan ingesteld worden door de juiste waarde aan register `eax` mee te geven. Zo betekent waarde `2` dat men geïnteresseerd is in de architectuur van de *cache*, zoals men kan zien aan de volgende *embedded assembly* code:

```
long     cache_eax, cache_ebx, cache_ecx, cache_edx;

asm("push %ebx;
     movl $2,%eax;
     CPUID;
     movl %eax,cache_eax;
     movl %ebx,cache_ebx;
     movl %ecx,cache_ecx;
     movl %edx,cache_edx;
     pop %ebx");
```

a. Waarom wordt alleen de oude waarde van register `ebx` op de *stack* veilig gesteld, terwijl er ook de gegevens in registers `eax, ecx` en `edx` worden overschreven?

b. De inhoud van de vier registers worden in vier globale variabelen opgeslagen. In welk gedeelte van het virtuele geheugen van een Linux-machine kan men globale variabelen vinden?



Student: **Collegekaartnummer:**

c. Deze 32-bytes variabelen blijken vier 8-bytes codes te bevatten die een onderdeel van de *cache* architectuur beschrijven. Voor een Pentium4 kan men de o.a. de volgende codes verwachten, met daarachter een beschrijving in tekst.

```
0x66: 1st-level data cache: 8K-bytes, 4-way set associative, sectored cache, 64-byte line size
0x70: Trace cache: 12K-uops, 8-way set associative
0x7B: 2nd-level cache: 512K-bytes, 8-way set associative, sectored cache, 64-byte line size
```

Heeft de Pentium4 een pure Von Neumann-architectuur?
Beargumenteer je antwoord.

d. Op een Linux-machine gaat men rekenen met op een kaart voor een robot, met de volgende *structure* voor de *grid-points*:

```
struct grid_point_s {
  char visited;
  int x,y;
  double occupied;
} *grid_point_p;
```

Wat is de grootte van deze *structure*?

e. Wat is de maximale waarde van `DIM` zodat de kaart `grid_point_s map[DIM][DIM]` nog net in de L2-cache van een Pentium4-machine past?

vraag 2

Bestudeer de volgende programma:

```
int val = 10;

void handler(sig)
{
      val += 5;
      return;
}

int main()
{
      int pid;

      signal(SIGCHLD, handler);

      if ((pid = fork()) == 0) {
            val -= 3;
            exit(0);
      }

      waitpid(pid, NULL, 0);

      printf("val = %d\n", val);

      exit(0);
}
```

Wat is de uitvoer van dit programma? `val =` _____

Leg uit waarom!

Student: **Collegekaartnummer:**

Vraag 3

U maakt deel uit van een programmeur's team dat probeert de snelste faculteit routine ter wereld te schrijven. Men was begonnen met een recursieve functie, maar men kwam er al snel achter dat een iteratieve versie veel sneller was:

```
int fact(int n)
{
    int i;
    int result = 1;
    for (i = n; i > 0; i--)
       result = result * i;
    return result;
}
```

Deze conversie reduceerde het aantal *cycles per element* (CPE) voor de functie van 63 naar 4. Echter, men is nog niet tevreden.

Één van de programmeurs had eens iets gehoord over *loop unrolling*. Deze programmeur probeerde dit toe te passen in de volgende versie:

```
int fact_u2(int n)
{
    int i;
    int result = 1;
    for (i = n; i > 0; i-=2)
           result = (result * i) * (i-1);
    return result;
}
```

Bij het testen kwam men erachter dat voor sommige waarden van *n* de nieuwe code 0 i.p.v. *n*! retourneert.

- **a.** Voor welke waarden van *n* verschillen de resultaten van functie van `fact` en `fact`_u2 van elkaar?
- **b.** Geef aan hoe men door een kleine verandering in de code wel het correcte resultaat kan verkrijgen.
- **c.** Metingen aan de routine `fact`_u2 geven nog steeds een CPE van 4.
  Kunt u dit verklaren?
- **d.** In een laatste poging verplaatst u de haakjes van de *inner loop* expressie.

  ```
  result = result * (i * (i-1));
  ```

  Plotseling heeft de *loop unrolling* wel effect, en daalt de CPE naar 2.5.
  Kunt u deze versnelling verklaren?

Student: **Collegekaartnummer:**

vraag 4

Deze vraag test jullie begrip van *stack frames*. Bekijk de volgende recursieve functie:

```
int silly(int n, int *p)
{
    int val, val2;

    if (n > 0)
        val2 = silly(n << 1, &val);
    else
        val = val2 = 0;

    *p = val + val2 + n;

    return val + val2;
}
```

Voor de betekenis van de IA32 instructies in het GAS-formaat kan men gebruik maken van de volgende tabel:

Instruction	**Effect | Jump condition**	**Description**
`movl S,D`	D ← S	Move double word
`pushl S`	R[%esp] ← R[%esp] - 4; M[R[%esp]] ← S	Push to stack
`popl S`	D ← M[R[%esp]] R[%esp] ← R[%esp] + 4;	Pop from stack
`leal S,D`	D ← &S	Load effective address
`xorl S,D`	D ← D ^ S	Exclusive-or
`incl D`	D ← D + 1	Increment
`decl D`	D ← D - 1	Decrement
`addl S,D`	D ← D + S	Add
`subl S,D`	D ← D - S	Subtract
`cmpl S2,S1`	S1 – S2	Compare double words
`testl S2,S1`	S1 & S2	Test double words
`jle Label`	(SF ^ OF) | ZF	Less or equal (signed <=)
`jg Label`	˜(SF ^ OF)& ˜ZF	Greater (signed >)
`call`		Procedure call
`ret`		Return from call

Hierbij staat S voor *Source*, D voor *Destination*, M voor *Memory*, R voor *Register*, CF voor *Carry Flag*, ZF voor *Zero Flag* en SF voor *Sign Flag*.

Student: **Collegekaartnummer:**

Deze functie wordt vertaald in de volgende assembly code:

```
silly:
pushl %ebp
movl %esp,%ebp
subl $20,%esp
pushl %ebx
movl 8(%ebp),%ebx
testl %ebx,%ebx
jle .L3
addl $-8,%esp
leal -4(%ebp),%eax
pushl %eax
leal (%ebx,%ebx),%eax
pushl %eax
call silly
jmp .L4
.p2align 4,,7
.L3:
xorl %eax,%eax
movl %eax,-4(%ebp)
.L4:
movl -4(%ebp),%edx
addl %eax,%edx
movl 12(%ebp),%eax
addl %edx,%ebx
movl %ebx,(%eax)
movl -24(%ebp),%ebx
movl %edx,%eax
movl %ebp,%esp
popl %ebp
ret
```

De gdb debugger laat na twee recursieve aanroepen de volgende informatie over de stack en het laatste stackframe zien:

```
(gdb) info stack
#0  silly (n=8, p=0xbfffec24) at silly.c:8
#1  0x0804842e in silly (n=4, p=0xbfffec54) at silly.c:8
#2  0x0804842e in silly (n=2, p=0xbfffec84) at silly.c:8
#3  0x0804846b in main () at silly.c:21

(gdb) info frame
Stack level 0, frame at 0xbfffec00:
 eip = 0x804841e in silly (silly.c:8); saved eip 0x804842e
 called by frame at 0xbfffec30
 source language c.
 Arglist at 0xbfffebf8, args: n=8, p=0xbfffec24
 Locals at 0xbfffebf8, Previous frame's sp is 0xbfffec00
 Saved registers:
  ebx at 0xbfffebe0, ebp at 0xbfffebf8, eip at 0xbfffebfc
```

De gdb debugger natuurlijk ook vragen waar de locale variabelen zijn opgeslagen:

```
(gdb) print &n
Address requested for identifier "n" which is in register $ebx
(gdb) print &p
Address requested for identifier "p" which is in register $eax
```

Student: **Collegekaartnummer:**

a. Wordt de variabele `val` in een register bewaard?
Zo ja, in welk register?
Zo nee, wat is dan de relatieve positie t.o.v. de *framepointer* `%ebp`?

b. Wordt de variabele `val2` in een register bewaard?
Zo ja, in welk register?
Zo nee, wat is dan de relatieve positie t.o.v. de *framepointer* `%ebp` ?

c. Één van de twee wordt op de stack bewaard. Waarom is dit noodzakelijk?

Succes!