Lösningsförslag

Systemnära Programmering, 5p
25 September 1992

- Maxpoäng 48 (för godkänt krävs 24p, väl godkänt 36)

Uppgift 1 (5 + 5 p)

(a)
```
POINTER STEW
-----
p1   ---
     p2 --
        p3---
          p4
```

(b)
```
pp-p = 1        *pp-a = 1       **pp = 8
pp-p = 2        *pp-a = 2       **pp = 12
pp-p = 3        *pp-a = 3       **pp = -3
pp-p = 3        *pp-a = 4       **pp = 4
```

Uppgift 2 (4 + 3 + 2 p)

(a)
```
double (*least(f1, f2, x))()
double (*f1)(), (*f2)(), x;
{
    if ((*f1)(x) > (*f2)(x)) /* eller f1(x) > f2(x) */
        return f2;
    else
        return f1;
}
```

eller med typedef så kan huvudet se ut så här:

```
typedef double (*FP)();
FP least(f1, f2, x)
FP f1, f2;
double x;
```

funktionskroppen blir densamma.

(b)
```
#define LEAST(f1,f2,x)  f1(x) > f2(x) ? f2 : f1
eller
#define LEAST(f1,f2,x)  (*f1)(x) > (*f2)(x) ? f2 : f1
```

(c) Macron är bara textsubstitution medan funktioner evaluerar sina parametrar. Eftersom macron inte är funktionsanrop så är de snabbare, bla på grund av att det inte behövs läggas upp någon aktiveringspost på stacken. Funktioner finns bara i en “kopia”, macron expanderas överallt.

De ger inte alltid samma resultat, tex

```
x = 0.0;
least(f1, f2, x++);
x = 0.0;
LEAST(f1, f2, x++);
```

least: anropar f1(0.0) och f2(0.0)

LEAST: anropar f1(1.0) och f2(0.0)

Dessutom får x värdet 1.0 respektive 2.0. Dessa båda egenheter beror på att parametern x förekommer två gånger i macro-delen, och eftersom det är ren textsubstitution så kommer $x++$ att förekomma två ggr.

Uppgift 3 (8 p)

```
#include    <stdio.h>
#include    <ctype.h>

#define DEFAULTSTEP 1

main(argc, argv)
int argc;
char *argv[];
{
int step = DEFAULTSTEP;
int c;

if (argc > 2) { /* Too many argument */
    fprintf(stderr, "usage: %s [-n]\n", argv[0]);
    fprintf(stderr, "n is the number of steps to circulate (0<=n<=26)\n");
    exit(1);
} else if (argc == 2) { /* Argument is the number of steps to circulate */
    if (sscanf(argv[1], "-%d", &step) != 1) {
        fprintf(stderr, "usage: %s [-n]\n", argv[0]);
        fprintf(stderr, "n is the number of steps to circulate (0<=n<=26)\n");
        exit(1);
    }
    if (step < 0 || step > 26) {
        fprintf(stderr, "usage: %s [-n]\n", argv[0]);
        fprintf(stderr, "n is the number of steps to circulate (0<=n<=26)\n");
        exit(1);
    }
}

while ((c = getchar()) != EOF) {
    if (isalpha(c))
        if (isupper(c))
            if (c + step > 'Z')
                c = 'A' + c + step - 'Z' - 1;
            else
                c += step;
        else
            if (c + step > 'z')
                c = 'a' + c + step - 'z' - 1;
            else
                c += step;
    putchar(c);
}
}
```

Uppgift 4 (3 + 3 + 1 p)

(a) **RUNNING:** exekverar instruktioner på processorn.

BLOCKED: väntar på något, tex I/O

READY: klar att exekvera, men processorn upptagen.

(b) **READY → RUNNING:** Processorn ledig och scheduleraren har valt processen.

READY ← RUNNING: Scheduleraren tycker att processen exekverat länge nog.

RUNNING → BLOCKED: Processen har exekverat en instruktion som innebär att den måste vänta på någonting, tex I/O.

BLOCKED ← READY Det processen har väntat på är klart, dvs processen är klar att exekvera när processorn blir ledig.

(c) De nya processerna kommer in till tillståndet READY.

Uppgift 5 (4 p)

En *short-term scheduler* är en algoritm som fördelar processortiden mellan olika processer.

En enkel algoritm är *First Come First Served* (FCFS) som helt enkelt tar den process som väntat längst i READY-kön och låter den exekvera så länge den vill, dvs tills den hamnar i BLOCKED eller blir klar. Då tar algoritmen nästa process i kön osv.

Fördelar: Enkel, snabb

Nackdelar: Kan bli långa väntetider om en lång process startas.

Uppgift 6 (4 + 6 p)

(a) Semaforer är ett synkroniseringsprimitiv, dvs de används för att synkronisera processer. Till exempel vid ömsesidig uteslutning av kritiska sektioner.

Semaforer består av en variabel och två operationer, signal och wait som är atomära. Dessutom kan semaforen initieras en gång.

wait: Om semaforens värde > 0 minska med ett och fortsätt exekvera, annars blockeras processen tills semaforens värde > 0. Bilda kö om flera processer blockeras.

signal: Om semaforens värde $<= 0$ väck en av de blockerade processerna, annars öka semaforens värde med ett.

(b)
```
monitor semaphore
{
   int initialized=FALSE, S;
   condition C;

   void init(value)
   int value;
   {
      if (initialized)
         monerror("init: Semaphore already initialized");
      S = value;
      initialized = TRUE;
   }

   void wait()
   {
      if (!initialized)
         monerror("wait: Semaphore is not initialized");
      if (S <= 0)
         delay(C);
      S--;
   }

   void signal()
   {
      if (!initialized)
         monerror("signal: Semaphore is not initialized");
      S++;
      continue(C);
   }
}
```

Så här skulle man kunna tänka sig pseudokod för en generell semafor implementerad med hjälp av en monitor. Monitorn garanterar att wait och signal blir atomära eftersom bara en process kan befinna sig i monitorn. Dessutom kan man endast ändra på S med wait och signal.

Användande: först `semaphore.init(1)` (binär semafor), sedan `semaphore.wait()` och `semaphore.signal()`.

Denna kod var alltså för en enda semafor, vill man ha fler kan man ersätta de tre variablerna med arrayer som rymmer önskat antal. Man måste då också skicka med en extra parameter till de 3 funktionerna (= index till arrayerna, ex `semaphore.wait(3)` $\Leftrightarrow$ wait på semafor nummer 3).