21.03.2013

Curs 05 - 4

- am pus regulile de calcul (mai simple)
- link-uri către screencapture → cum să programăm linia de comandă
- grep / sed / awk + C language ; o linie de comandă sau program awk

singurele instrucțiuni ~~pot fi într~~ care nu pot fi întrerupte sunt cele din ASM
nu pot fi întrerupt

$. echo 0 > a.txt

vi ~~a.txt 6.sh~~ inc.sh

```
#!/bin/bash
F=$1
N=0
while [ $N -lt 300 ]; do
        K=`cat ~~a.txt~~ $F`
        K=`expr $K +1`
        echo $K > $F
        N=`expr $N +1`
done
```

./inc.sh a.txt

300

echo 0 > a.txt

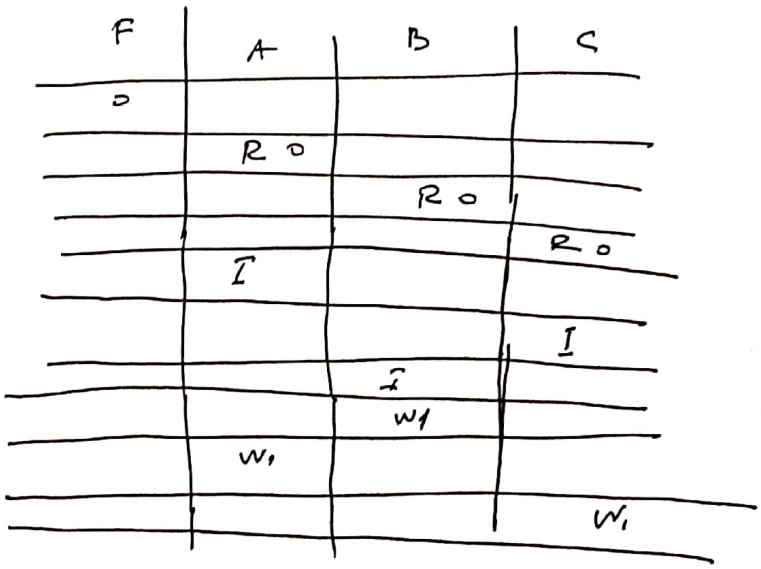
./inc.sh a.txt în două locuri deodată → comportament nedefinit (595, 456, 538)

R - citire
I - incrementare
W - scriere

vi run.sh

```
#!/bin/bash
echo 0 > a.txt
./inc.sh a.txt &
./inc.sh a.txt &
./inc.sh a.txt &
./inc.sh a.txt
```

chmod 700 run.sh



→ dacă nu se rulează singure, programul ~~nu~~ nu funcționează
→ sistemul decide când să întrerupă procesele
→ programele nu se rulează singure
→ procesele nu rulează în paralel, ele de fapt sunt întrerupte foarte des
→ apar erori pe care nu le puteți vedea, genul de eroare în care se blochează random deși în mod normal funcționa

vi inc.c

```
#include <stdio.h>
int main (int argc , char **argv) {
    int i, k, f;
    FILE *f;
    f = fopen("argv[1]");
    f = open ( argv[1] , O_RDWR);
    k = 0; write(f, &k, sizeof(int));
    for (i=0 ; i < 256; i++) {
        lseek (f, 0, SEEK_SET);
        read (f, &k, sizeof(int)) ;
        k++;
        write f
        lseek (f, 0 , SEEK_SET);
        write (f, &k, sizeof(int));
    }
    close(f);
```

if (argc==2 && strcmp(argv[2], "reset")==0) { k=0; write(f, &k, sizeof(int)); close(f); return 0; }

---

gcc -Wall -g -o inc inc.c

---

cat /dev/null > b.dat
./inc 6.dat reset
xxd 6.dat
./inc 6.dat
valgrind ./inc 6.dat

---

6. run-c.sh

./inc 6-dat reset
./inc 6.dat &
./inc 6.dat &
./inc 6.dat

→ au o resursă comună => behaves weird

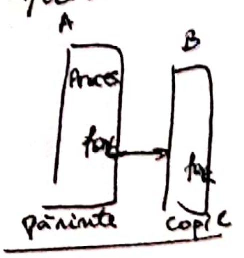
resursă critică - orice resursă care este accesată de mai multe procese / threaduri dintre care cel puțin unul o modifică

regiune critică - zonă de cod care accesează resursa critică

race condition

what's sleep

fork




```
vi fork-a.c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char** argv) {
    printf("before fork. \n");
    fork();
    printf("Afterfork. \n");
    sleep (10);
    printf("After sleep \n");
}   return 0;
```

→ Before fork
After fork
After fork
After sleep
After sleep

```
printf("before fork\n");
sleep (5)
~~rotate~~
```

fork → copiază procesul părinte în procesul fiu
→ returnează pid-ul procesului fiu în părinte și 0 în fiu

```
sys/types.h
sys/wait.h
int r;
r = fork();
printf ("Proces = %d  Parinte=%d   R=%d \n", getpid(), getppid(), r);
wait(0);
return 0;
```

```
int pid;
pid = fork();
if (pid == 0) {
    printf("Proces fiu \n");   ~~exit(0);~~
}
printf ("Proces părinte \n");
return 0;
```

⟹ Proces parinte
Proces fiu
Proces parinte

```
int pid;
pid = fork();
if (pid == 0) {
    printf("Proces fiu \n");
}   exit(0); // omoară procesul, oriunde s-ar afla
printf("Proces părinte \n");
return 0;
```

```
while (1) {
    read req
    process req
    write resp
}
```

SERVER NAȘPA

```
while (1) {
    read req
    if (fork() == 0) {
        process req
        write resp
        exit(0);
    }
}
```

SERVER MAI BLÂNZIȚĂ

→ se păstrează procesul fiu într-o stare zombie pentru că procesul părinte care trebuie să exit code

wait(0) → se ia exit code de la procesul fiu și e nu mai e zombie

```
killall
while (1) {
    fork();
}
```

```
int pid;
pid = fork();
if (pid == 0) {
    sleep(5);
    exit(0);
}
sleep(10);
return 0;
```

↳ generates zombi

```
int pid;
pid = fork();
if (pid == 0) {
    exit(0);
}
wait(0); // ← părinților nu se termină
         // până nu ia exit code
         // parametrul de la wait e un pointer spre int
```

↳ wait pid

```
for (i=0; i<3; i++) {
    fork();
}
```

```
for (i=0; i<3; i++) {
    pid = fork();
    if (pid == 0) {
        exit(0);
    }
}
for (i=0; i<3; i++) {
    wait(0);
}
```

→ creează doar 3 procese fiu

i = 0

i = 1

i = 2