Universidade da Beira Interior
Departamento de Informática

Exercise 2 Report

Elaborated by:

Joel Tapia a47275
Manuel Garcia a45500
Tiago Ribeiro a46346

June 3, 2023

Chapter

1

Intro

1.1 Motivation

This project is made in the scope of the Sistemas Operativos, a **UC!** (**UC!**) Sistemas operativos, of the second year of the Informatic Engineering course in **UBI!** (**UBI!**).

The program tries to simulate (using pipes) a tokenring that stops incrementing its token value when a certain condition is met.

We implement two conditions with similar outcomes, the first one is when the token becomes as big as the last argument passed to the program through linux bash, once that happens, the program stops itself it is worth to mention that two processes occur in parallel during the life of the tokenring program.

The second outcome happens when we press control+c and the event exit is handled by two functions . Leaving the user in the know of the program's termination.

1.2 Goals

The goals that we were trying to reach with the implementation of the tokenring were the following:

- To make a tokenring that increments the token as long as its value (the token's) remains smaller than max. Every time the token is incremented a number is generated, if this number is bigger than the probability passed as parameter times a hundred the program will showcase a message letting the user know how the program is doing.
- The same as above, but the stop condition will now be pressing CTRL+C.

Chapter

2

Desenvolvimento e implementação

2.1 Descrição do código

The code begins by receiving the bash values and passing them to variables in both versions there are 2 integers and one float, but in the first version there is an additional integer, who is the max value that the token could reach. The CPU then proceeds to create pipes that will receive and send values from one process to another.

There is "n" forks, when the forks happens, we create two processes, one that will print things to the bash and another for implementing the meat of the algorithm.

For the duration of this report, I will refer to the printing process as the client and to the "hidden" one as the server.

```
1    int numDePipetas= atoi(argv[1]), max = atoi(argv[4]), ↩
         tempoDeEspera= atoi(argv[3]);
2    float prob= atof(argv[2]);
```

The first thing that the program will do after declaring the variables is create the pipes that would have this type of name "pipeto", just after this the algorithm begins.

Also the process are built in this part because of the cycle "for" which is repeated until "n", in each one the fork function is called and in the child process the pipes that we created above are opened following this form, the pipe "n" will be open for reading and the pipe "n+1" will be just for writing, this is a cycle until the max pipe are reached, only here the last one is going to connect with the first and the algorithm repeat.

```
1    for(int i = 0; i< numDePipetas; i++){
2        char * buffer = (char *) malloc(15);
3        if(i!=numDePipetas-1){
4            sprintf(buffer,"pipe%dto%d",i+1,i+2);
5        }else  sprintf(buffer,"pipe%dto%d",numDePipetas,1);
6        mkfifo(buffer,PERMS);
7        strncpy(nomes[i],buffer,15);
8        memset(buffer,0,15);
9        free(buffer);
10
11   for(int i = 0; i< numDePipetas  ;i ++){
12       pid = fork();
13       if(pid == 0){ // processo filho
14           ids[i] = getpid();
15           unsigned int seed = ids[i];
16           srand(seed);
17           //abrir pipes
18
19           pipetas[i][0] = open(nomes[i], O_RDONLY);
20           pipetas[i][1] = open(nomes[(i + 1) % numDePipetas],↩
                 O_WRONLY);
21   }
```

The server does all the manipulation of data needed in the program, it creates named pipes for the execution, and writes to them, so when the client part hits a read() function it must wait for the write() function that only happens in the server. The function dictates whether we print the current state to the screen or not based on chance that is as big as the user wants it to be. Since the write() always occurs before read() the else statement reaches its end before the if-clause much sooner, luckily one of the C libraries provides us with the wait() function avoiding a premature return.

```
1    else{
2        mknod(FIFO1, S_IFIFO | PERMS, 0);
3        pipe1[1] = open(FIFO1,1);
4        int pipetasDeEscrita[numDePipetas ][2];
5        for(int i = 0;i<numDePipetas  ;i++){
6            char * bufferEsc = malloc(15);
7            sprintf(bufferEsc,"/tmp/f%d",i+2);
8            //limpar
9            mknod(bufferEsc,S_IFIFO | PERMS ,0);
10           pipetasDeEscrita[i][1] = open(bufferEsc,1);
11           memset(bufferEsc,0,15);
12           free(bufferEsc);
13       }
14       int token=0;
```

```
15          int i=0;
16          while (token<= max){
17              if(token == max)
18              {
19                  //printf("FIM\n");
20                  write(pipe1[1],&token,sizeof(int) );
21                  break;
22              }
23              if(((int) (prob * 100)) >= (rand() % 100 + 1)){
24                  //printf("OCORREU, %d\n", i);
25                  write(pipe1[1],&token,sizeof(int));
26                  write(pipetasDeEscrita[i][1],&(i),sizeof(int));
27
28              }
29              if(i== numDePipetas- 1){
30                  i=0;
31              }
32              else{
33                  i++;
34              }
35
36              token++;
37          }
38          close(pipe1[1]);
39          for(int i = 0; i<numDePipetas ; i++){
40              close(pipetasDeEscrita[i][1]);
41          }
42
43          wait(NULL);
44
45      }
```

2.2 Código fonte

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <sys/wait.h>
5  #include <fcntl.h>
6  #include <semaphore.h>
7  #include <unistd.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <time.h>
11 #include <signal.h>
```

```
12 #define PERMS 0666
13 #define DEBUGGING 0
14 #define TRUE 1
15
16 int token = -1;
17 int parent;
18 int *ids;
19 int ** pipetas;
20 volatile char lock =0;
21
22 sem_t sem;
23
24 void trancar(){
25     label:{
26         char local = 1;
27         char tmp = lock;
28         lock = local;
29         local =tmp;
30
31         if(0 != local){
32             goto label;
33         }
34     }
35 }
36
37 void destrancar(){
38     lock = 0;
39 }
40
41 int main (int argc,char * argv[] )
42 {
43
44     //signal(SIGINT, stopHandler);
45     parent = getpid();
46     int numDePipetas= atoi(argv[1]), max = atoi(argv[4]), ↩
           tempoDeEspera= atoi(argv[3]);
47
48     float prob= atof(argv[2]);
49     int probInt = (int) (100 * prob);
50     int pid   = -1;
51     ids= malloc(numDePipetas * sizeof(int));
52     char nomes[numDePipetas][15];
53     int len = sizeof(int *) * numDePipetas + sizeof(int) * 2 * ↩
           numDePipetas;
54     pipetas = (int **) malloc(len);
55     int * ptr = ( int *) (pipetas+numDePipetas);
56
57     for(int i = 0; i< numDePipetas; i++){
58         pipetas[i] = (ptr + 2 * i);
```

```
59      }
60
61      for(int i = 0; i< numDePipetas; i++){
62          char * buffer = (char *) malloc(15);
63
64          if(i!=numDePipetas-1){
65              sprintf(buffer,"pipe%dto%d",i+1,i+2);
66          }else  sprintf(buffer,"pipe%dto%d",numDePipetas,1);
67
68          mkfifo(buffer,PERMS);
69          strncpy(nomes[i],buffer,15);
70          memset(buffer,0,15);
71          free(buffer);
72
73          if(DEBUGGING){ printf("CRIAR PIPE\n");}
74      }
75
76      for(int i = 0; i< numDePipetas  ;i ++){
77          pid = fork();
78          if(pid == 0){ // processo filho
79              if(DEBUGGING){ printf("CRIAR FILHO\n");}
80              ids[i] = getpid();
81              unsigned int seed = ids[i];
82              srand(seed);
83              if(DEBUGGING){ printf("STACK?\n");}
84              //abrir pipes
85              if(i==numDePipetas-1){
86                  pipetas[i][0] = open(nomes[i], O_RDONLY);
87                  pipetas[i][1] = open(nomes[0],O_WRONLY);
88              }
89              else{
90                  pipetas[i][0] = open(nomes[i], O_RDONLY);
91                  pipetas[i][1] = open(nomes[i+1],O_WRONLY);
92              }
93
94
95
96              if(DEBUGGING){ printf("STACK?\n");}
97
98              while(1){
99                  read(pipetas[i][0], &token, sizeof(int));
100                 // printf("[p%d] received token (val = %d)\n", i + 1, ↩
                        token);
101                 if(token >= max){
102                     for(int k=0;k<numDePipetas;k++){
103                         close(pipetas[k][1]);
104                         close(pipetas[k][0]);
105                         unlink(nomes[k]);
106                     }
```

```
107
108                     for(int k=0;k<numDePipetas;k++ ){
109                         if(k!=i){kill(ids[k],SIGINT);}
110                     }
111                     fflush(stdout);
112                     return 0;
113                 }
114
115
116
117                 if((rand() % 100 + 1 ) < probInt){
118                     printf("[p%d] blocked on token (val = %d) PROCESSO↩
                            = %d\n",i+1,token, getpid());
119                     sleep(tempoDeEspera);
120                     fflush(stdout);
121                 }
122
123                 if(token >= max){
124                     for(int k=0;k<numDePipetas;k++){
125                         close(pipetas[k][1]);
126                         close(pipetas[k][0]);
127                         unlink(nomes[k]);
128                     }
129
130                     for(int k=0;k<numDePipetas;k++ ){
131                         if(k!=i){kill(ids[k],SIGINT);}
132                     }
133                     fflush(stdout);
134                     return 0;
135                 }
136
137                 int novoToken = token + 1;
138                 write(pipetas[i][1], &novoToken, sizeof(int));
139             }
140
141         }
142
143     }
144
145     int fd_write = open("pipe1to2", O_WRONLY);
146     token = 0;
147     write(fd_write, &token, sizeof(int));
148
149     for(int i = 0; i < numDePipetas; i++){
150         wait(NULL);
151     }
152
153     return !TRUE;
154 }
```

Excerto de Código 2.1: Exercise 2 source code part 1.

```
#include<stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <signal.h>
#define PERMS 0666
#define DEBUGGING 0
#define TRUE 1

int token = -1;
int parent, numpip;
int *ids;
int ** pipetas;
char **nomes;
volatile char lock =0;

sem_t sem;

void stopHandler(int signum){
    int pid = getpid();
    for(int k=0;k<numpip;k++){
        close(pipetas[k][1]);
        close(pipetas[k][0]);
        unlink(nomes[k]);
    }
    for(int k=0;k<numpip;k++ ){
        if(pid != ids[k]){
           kill(ids[k],SIGINT);
        }
    }

    raise(SIGINT);
    exit(0);

}

void trancar(){
    label:{
```

```
45          char local = 1;
46          char tmp = lock;
47          lock = local;
48          local =tmp;
49
50          if(0 != local){
51              goto label;
52          }
53      }
54  }
55
56  void destrancar(){
57      lock = 0;
58  }
59
60  int main (int argc,char * argv[] )
61  {
62      sem_init(&sem,0,1);
63
64      parent = getpid();
65      int numDePipetas= atoi(argv[1]), tempoDeEspera= atoi(argv[3]);
66      numpip = numDePipetas;
67      float prob= atof(argv[2]);
68      int probInt = (int) (100 * prob);
69      int pid   = -1;
70      ids= malloc(numDePipetas * sizeof(int));
71      int tamanho = sizeof(char *) * numDePipetas + sizeof(char) * 15 * ↩
            numDePipetas;
72      nomes = (char**) malloc(tamanho);
73      char * apontador = (char *) (nomes + numDePipetas);
74
75      for(int i = 0; i< numDePipetas; i++)
76      {
77          nomes[i] = (apontador + 15 * i);
78      }
79
80      int len = sizeof(int *) * numDePipetas + sizeof(int) * 2 * ↩
            numDePipetas;
81      pipetas = (int **) malloc(len);
82      int * ptr = ( int *) (pipetas+numDePipetas);
83
84      for(int i = 0; i< numDePipetas; i++)
85      {
86          pipetas[i] = (ptr + 2 * i);
87      }
88
89      for(int i = 0; i< numDePipetas; i++){
90          char * buffer = (char *) malloc(15);
91          if(i!=numDePipetas-1){
```

```
92          sprintf(buffer,"pipe%dto%d",i+1,i+2);}
93          else  sprintf(buffer,"pipe%dto%d",numDePipetas,1);
94          mkfifo(buffer,PERMS);
95          strncpy(nomes[i],buffer,15);
96          memset(buffer,0,15);
97          free(buffer);
98          if(DEBUGGING){ printf("CRIAR PIPE\n");}
99      }
100
101     for(int i = 0; i< numDePipetas  ;i ++){
102         pid = fork();
103         signal(SIGINT, stopHandler);
104
105         if(pid == 0){ // processo filho
106             if(DEBUGGING){ printf("CRIAR FILHO\n");}
107             ids[i] = getpid();
108             unsigned int seed = ids[i];
109             srand(seed);
110             if(DEBUGGING){ printf("STACK?\n");}
111             //abrir pipes
112
113             pipetas[i][0] = open(nomes[i], O_RDONLY);
114             pipetas[i][1] = open(nomes[(i + 1) % numDePipetas],↩
                    O_WRONLY);
115
116
117             if(DEBUGGING){ printf("STACK?\n");}
118
119             while(1){
120
121                 read(pipetas[i][0], &token, sizeof(int));
122                 if((rand() % 100 + 1 ) < probInt){
123                     printf("[p%d] blocked on token (val = %d) PROCESSO↩
                            = %d\n",i+1,token, getpid());
124                     sleep(tempoDeEspera);
125                 }
126
127                 int novoToken = token + 1;
128                 write(pipetas[i][1], &novoToken, sizeof(int));
129             }
130
131         }
132
133     }
134
135
136     int fd_write = open("pipe1to2", O_WRONLY);
137     token = 0;
138     write(fd_write, &token, sizeof(int));
```

```
139
140     for(int i = 0; i < numDePipetas; i++){
141         wait(NULL);
142     }
143
144     return !TRUE;
145 }
```

Excerto de Código 2.2: Exercise 2 source code part 2.

Chapter

3

Examples of execution

3.1 First program TokenRing

For this program we need four inputs that made the program work correctly, the first input is "n" that means the number of channels currently creates. The second one is "p" the propability of the token to be blocked and also print the actual number of the token. Third number "t" is the timer between executions. The last one "m" will define the maximun number of executions and the max value of the token.

Below is an example of the program's output that prints every time that the token is blocked, it has to show the number of channel and the value of the token.

```
darwin@Darwincsg:~/SistemasOperativos/TrabalhoemGrupo$ ./tokenring 5 0.4 4 50
[p1] blocked on token (val = 5)
[p5] blocked on token (val = 9)
[p2] blocked on token (val = 11)
[p1] blocked on token (val = 15)
[p5] blocked on token (val = 24)
[p1] blocked on token (val = 25)
[p2] blocked on token (val = 26)
[p1] blocked on token (val = 30)
[p3] blocked on token (val = 32)
[p4] blocked on token (val = 33)
[p1] blocked on token (val = 35)
[p4] blocked on token (val = 43)
[p5] blocked on token (val = 44)
[p2] blocked on token (val = 46)
[p4] blocked on token (val = 48)
darwin@Darwincsg:~/SistemasOperativos/TrabalhoemGrupo$
```

3.2 Second program Tokenring

In this second program we just need 3 values for it's correct execution, the third values have the same function but here the value "m" doesn't exist so the program will run until we terminated it, in this cases the example below we finish the execution with "ctrl-C".

```
darwin@Darwincsg:~/SistemasOperativos/TrabalhoemGrupo$ ./tokensol3 5 0.4 3
[p2] blocked on token (val = 1)
[p3] blocked on token (val = 2)
[p5] blocked on token (val = 4)
[p2] blocked on token (val = 6)
[p3] blocked on token (val = 7)
[p4] blocked on token (val = 8)
[p5] blocked on token (val = 9)
[p3] blocked on token (val = 12)
[p4] blocked on token (val = 13)
[p1] blocked on token (val = 15)
[p2] blocked on token (val = 16)
^CTerminating...
darwin@Darwincsg:~/SistemasOperativos/TrabalhoemGrupo$
```