Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Λειτουργικά Συστήματα – Άσκηση 3

Των Προπτυχιακών Φοιτητών
Αρώνη Παναγιώτη Α.Μ. 03109083
Δανασή Παναγιώτη Α.Μ. 03109004
e-mail: el09004@mail.ntua.gr
panaro32@hotmail.com

1 Ασκήσεις:

1.1 Υλοποίηση σημαφόρων με σωληνώσεις του UNIX:

Ερωτήσεις:

1. Το pipesem-test.c δημιουργεί μία διεργασία παιδί που κοιμάται για 5 δευτερόλεπτα και έπειτα τερματίζει. Τρέχουν δύο διεργασίες, η διεργασία γονέας και η διεργασία παιδί. Ο συγχρονισμός γίνεται με χρήση σημαφόρων. Συγκεκριμένα ο πατέρας κάνει pipesem_wait(&sem) και με αυτόν τον τρόπο μπλοκάρει την λειτουργία του μέχρι να έρθει το παιδί και να γράψει στον σημαφόρο με την εντολή pipesem_signal(sem). Αυτό μας εξασφαλίζει ότι τα δύο τελευταία μηνύματα θα τυπωθούν με την σωστή σειρά και συγκεκριμένα θα γίνουν ως εξής:
   printf("Child: signaling semaphore\n");
   printf("Parent: signaled, program should terminate.\n");
   (Τα δύο πρώτα μηνύματα που τυπώνουν οι διεργασίες αυτές μπορούν να τυπωθούν με οποιαδήποτε σειρά ανάλογα με την χρονοδρομολόγηση των διεργασιών.)

2. Η βιβλιοθήκη pipesem.c για να υλοποιήσει τους σημαφόρους κάνει χρήση των σωληνώσεων του UNIX. Συγκεκριμένα δημιουργείται για κάθε σημαφόρο 1 pipe στο οποίο γράφουμε τόσους ακεραίους όση και η τιμή στην οποία θέλουμε να αρχικοποιήσουμε τον σημαφόρο μας. Η λειτουργία wait υλοποιείται με το να διαβάζουμε από το pipe ενώ η λειτουργία signal με το να γράφουμε σε αυτό. Η χρήση των pipes μας εξασφαλίζει ότι η πράξη wait σε σημαφόρο μηδενικής τιμής (αρνητική τιμή σε pipe δεν μπορούμε να έχουμε) θα μπλοκάρει, μια και η πράξη read σε άδειο pipe μπλοκάρει την διεργασία μέχρι κάποιος να γράψει στο pipe.

3. Ο συγκεκριμένος τρόπος υλοποίησης θα μπορούσε να οδηγήσει σε λιμοκτονία αν η επιλογή των διεργασιών που θα διαβάσουν από το pipe γινόταν με μη δίκαιο τρόπο. Για παράδειγμα, αν το ΛΣ επέλεγε να διαβάζουν οι διεργασίες με μεγαλύτερη προτεραιότητα, τότε διεργασίες με μικρή προτεραιότητα μπορεί να οδηγούνταν σε λιμοκτονία. Συνεπώς ο τρόπος που επιλέγει το ΛΣ τις διεργασίες που κάθε φορά θα διαβάσουν από το pipe επηρεάζει άμεσα και την λειτουργία των σημαφόρων μας σε αυτήν την υλοποίηση.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "pipesem.h"

void pipesem_init(struct pipesem *sem,int val)
{
    int pfd[2],data=42;
    if(pipe(pfd)<0)
    {
        perror("pipesem_init: pipe");
        exit(1);
    }
    sem->rfd=pfd[0];
    sem->wfd=pfd[1];
    for(;val>0;val--)
        if(write(sem->wfd,&data,sizeof(int))<0)
        {
            perror("pipesem_init: write");
            exit(1);
        }
}

void pipesem_wait(struct pipesem *sem)
{
    int data;
    if(read(sem->rfd,&data,sizeof(int))<0)
    {
        perror("pipesem_wait: read");
        exit(1);
    }
}

void pipesem_signal(struct pipesem *sem)
{
    int data=42;
    if(write(sem->wfd,&data,sizeof(int))<0)
    {
        perror("pipesem_signal: write");
        exit(1);
    }
}

void pipesem_destroy(struct pipesem *sem)
{
    if(close(sem->rfd)<0)
    {
        perror("pipesem_destroy: close");
        exit(1);
    }
    if(close(sem->wfd)<0)
    {
        perror("pipesem_destroy: close");
        exit(1);
    }
}
```

Έξοδος εκτέλεσης προγράμματος:

Parent: waiting on semaphore

Child: sleeping for five seconds

Child: signaling semaphore

Parent: signaled, program should terminate.

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot:

Ερωτήσεις:

1. Για το σχήμα συγχρονισμού που υλοποιήσαμε χρειάζονται N σημαφόροι (όσοι και οι διεργασίες στις οποίες μοιράζεται ο υπολογισμός).

2. Για τον σειριακό υπολογισμό έχουμε: real 0m1.187s
   user 0m1.032s
   sys 0m0.084s
   Για τον παράλληλο υπολογισμό έχουμε: real 0m1.067s
   user 0m1.004s
   sys 0m0.060s

3. Το πρόγραμμά μας σε παράλληλο υπολογισμό εμφανίζει επιτάχυνση. Αυτό συμβαίνει γιατί στο κρίσιμο τμήμα έχουμε τοποθετήσει μόνο την εκτύπωση της κάθε γραμμής και όχι τον υπολογισμό της. Αν είχαμε τοποθετήσει και τον υπολογισμό, τότε θα ήταν σαν να τρέχαμε το πρόγραμμα σειριακά (ίσως και χειρότερα καθώς έχουμε και τον χρόνο του context switch) καθώς για να υπολογίσει κάθε διεργασία τα δεδομένα που πρέπει να τυπώσει, θα έπρεπε να περιμένει την προηγούμενη να τελειώσει. Αντίθετα, στην δικιά μας υλοποίηση ο υπολογισμός γίνεται παράλληλα και φροντίζουμε στο κρίσιμο τμήμα να είναι μόνο το τύπωμα ώστε να εμφανίζονται οι γραμμές με την σωστή σειρά.

4. Αν πατήσουμε CTRL+C την ώρα που εκτελείται το πρόγραμμά μας, τότε του στέλνουμε το σήμα SIGINT και το πρόγραμμά μας πεθαίνει. Αυτό έχει ως αποτέλεσμα το χρώμα των γραμμάτων στο τερματικό να παραμείνει στο τελευταίο χρώμα που έχει επιλεγεί για τύπωμα πριν διακόψουμε το πρόγραμμα μας. Για να εξασφαλίσουμε ότι το τερματικό θα γυρνάει στην προηγούμενη κατάστασή του σε μία τέτοια περίπτωση θα πρέπει να φροντίσουμε μέσα στο mandel.c να πιάνουμε το σήμα SIGINT και στην συνάρτηση χειρισμού του σήματος να επαναφέρουμε το χρώμα στην αρχική του κατάσταση και μετά να τερματίζουμε το πρόγραμμά μας.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <math.h>

#include "pipesem.h"
#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

// Number of processes to create
int procs_no=2;

// Output at the terminal is x_chars wide by y_chars long
int y_chars=50;
int x_chars=90;

// The part of the complex plane to be drawn:
// upper left corner is (xmin,ymax), lower right corner is (xmax,ymin)
double xmin=-1.8,xmax=1.0;
double ymin=-1.0,ymax=1.0;

// Every character in the final output is
// xstep x ystep units wide on the complex plane.
double xstep;
double ystep;

// functions
void child_proc(struct pipesem*,struct pipesem*,int);
void command_line_arguments(int,char*[]);
void compute_mandel_line(int,int[]);
void output_mandel_line(int,int[]);

int main(int argc,char *argv[])
{
    int cnt,status;
    // process command line arguments
    command_line_arguments(argc,argv);
    // calculate steps
    xstep=(xmax-xmin)/x_chars;
    ystep=(ymax-ymin)/y_chars;
    // initialize a semaphore for every child process
    struct pipesem *sems[procs_no];
    for(cnt=0;cnt<procs_no;cnt++) sems[cnt]=(struct pipesem*)malloc(sizeof(struct pipesem));
    for(cnt=0;cnt<procs_no;cnt++) pipesem_init(sems[cnt],0);
    // signal the first child process
    pipesem_signal(sems[0]);
    // fork every child process
    pid_t pids[procs_no];
    for(cnt=0;cnt<procs_no;cnt++)
    {
        pids[cnt]=fork();
        if(pids[cnt]<0)
        {
            perror("main: fork");
```

```
            exit(1);
        }
        if(!pids[cnt])
        {
            // child process
            child_proc(sems[cnt],sems[(cnt+1)%procs_no],cnt);
            exit(1);
        }
    }
    // wait for all child processes to exit
    for(cnt=0;cnt<procs_no;cnt++) wait(&status);
    // destroy the semaphores
    for(cnt=0;cnt<procs_no;cnt++) pipesem_destroy(sems[cnt]);
    // set terminal color back to default
    reset_xterm_color(1);
    return 0;
}

// Computes and outputs the mandel lines such that: line % procs_no == line_no
void child_proc(struct pipesem *sem,struct pipesem *next_sem,int line_no)
{
    int color_val[x_chars],line;
    for(line=line_no;line<y_chars;line+=procs_no)
    {
        compute_mandel_line(line,color_val);
        pipesem_wait(sem);
        output_mandel_line(STDOUT_FILENO,color_val);
        pipesem_signal(next_sem);
    }
    exit(0);
}

// Processing command line arguments given to customize the output
void command_line_arguments(int argc,char *argv[])
{
    int cnt; char opt;
    for(cnt=1;cnt<argc;cnt++)
    {
        if(strlen(argv[cnt])<3||argv[cnt][1]!='=')
        {
            printf("Invalid command line arguments: Wrong syntax\n");
            printf("Check syntax: [option_character]=[value_to_set]\n");
            exit(1);
        }
        opt=argv[cnt][0];
        argv[cnt][0]=argv[cnt][1]=' ';
        switch(opt)
        {
            case 'n': procs_no=atoi(argv[cnt]); break;
            case 'N': procs_no=atoi(argv[cnt]); break;
            case 'w': x_chars=atoi(argv[cnt]); break;
            case 'W': x_chars=atoi(argv[cnt]); break;
            case 'l': y_chars=atoi(argv[cnt]); break;
            case 'L': y_chars=atoi(argv[cnt]); break;
            case 'x': xmin=atof(argv[cnt]); break;
            case 'X': xmax=atof(argv[cnt]); break;
            case 'y': ymin=atof(argv[cnt]); break;
            case 'Y': ymax=atof(argv[cnt]); break;
            default : printf("Invalid command line arguments: Invalid option character\n");
```

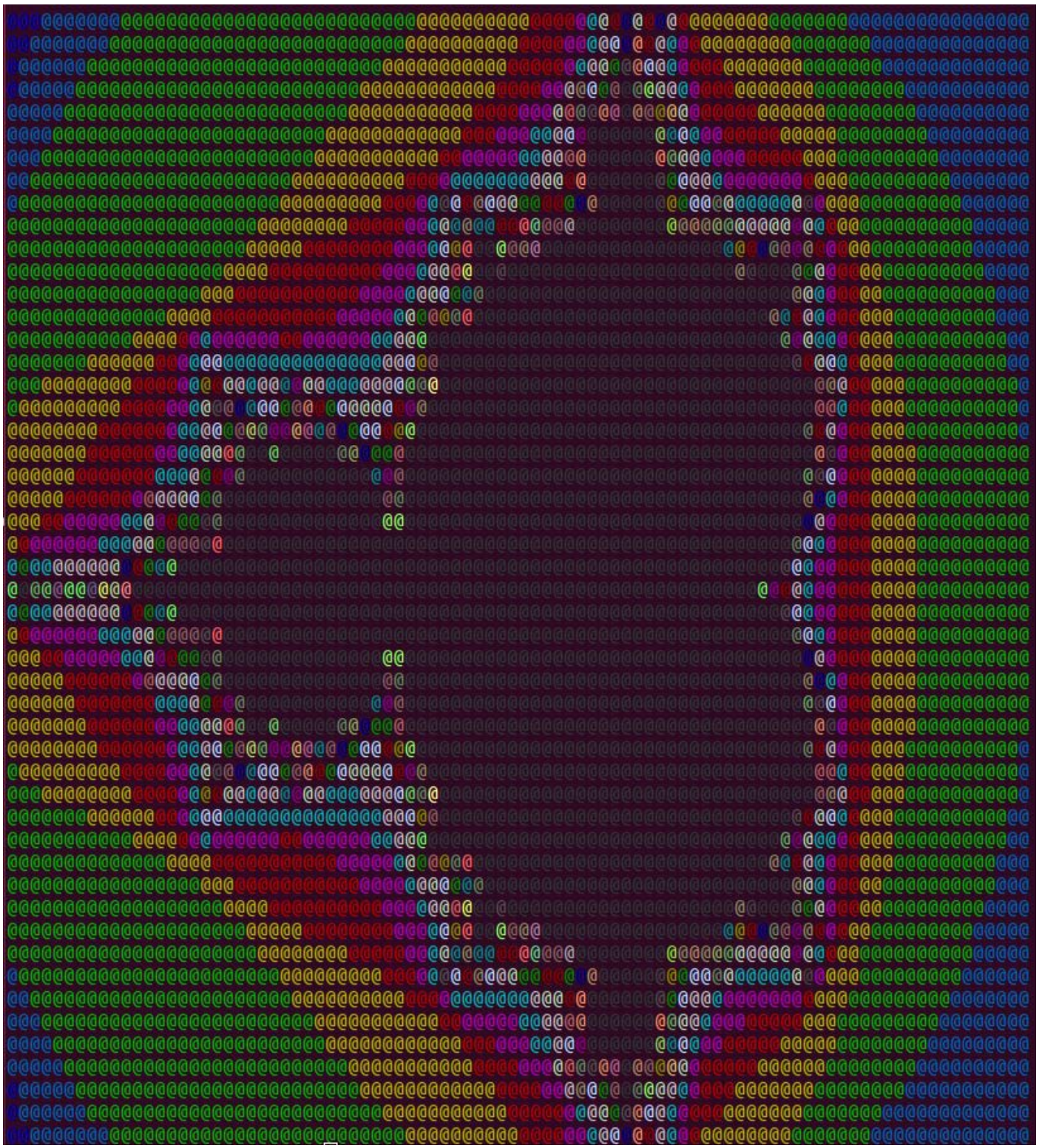
```
                    printf("Check options: n/N, w/W, l/L, x,X, y,Y\n");
                    exit(1);
        }
    }
    printf("\nProcesses(n/N): %d\nScreen(w/W*l/L): %d*%d\n",procs_no,x_chars,y_chars);
    printf("Plot([x,X]*[y,Y]): [%.3f,%.3f]*[%.3f,%.3f]\n\n\n",xmin,xmax,ymin,ymax);
    if(procs_no<=0||x_chars<=0||y_chars<=0||xmin>=xmax||ymin>=ymax)
    {
        printf("Invalid command line arguments: Wrong values\n");
        printf("Check values: N>0, W>0, L>0, x<X, y<Y\n");
        exit(1);
    }
}

// This function computes a line of output
// as an array of x_char color values.
void compute_mandel_line(int line,int color_val[])
{
    // x and y traverse the complex plane.
    double x,y;
    int n,val;
    // Find out the y value corresponding to this line
    y=ymax-ystep*line;
    // and iterate for all points on this line
    for(x=xmin,n=0;x<=xmax;x+=xstep,n++)
    {
        // Compute the point's color value
        val=mandel_iterations_at_point(x,y,MANDEL_MAX_ITERATION);
        if(val>255) val=255;
        // And store it in the color_val[] array
        val=xterm_color(val);
        color_val[n]=val;
    }
}

// This function outputs an array of x_char color values
// to a 256-color xterm.
void output_mandel_line(int fd,int color_val[])
{
    int i;
    char point='@';
    //char point[3]={0xe2,0x96,0x88};
    char newline='\n';
    for(i=0;i<x_chars;i++)
    {
        // Set the current color, then output the point
        set_xterm_color(fd,color_val[i]);
        if(write(fd,&point,1)!=1)
        {
            perror("output_mandel_line: write point");
            exit(1);
        }
    }
    // Now that the line is done, output a newline character
    if(write(fd,&newline,1)!=1)
    {
        perror("output_mandel_line: write newline");
        exit(1);
    }
```

```
}
```

Έξοδος εκτέλεσης προγράμματος:



1.3 Ταυτόχρονη πρόσβαση σε μοιραζόμενους πόρους:

Ερωτήσεις:

1. Η create_shared_memory_area δημιουργεί ένα κομμάτι μοιραζόμενης μνήμης όπου εκεί περιέχεται η μεταβλητή n την οποία και μεταβάλουν οι διεργασίες μας.

2. Το σχήμα συγχρονισμού που έχουμε υλοποιήσει χρησιμοποιεί ένα σημαφόρο για κάθε διεργασία και λειτουργεί ως εξής: Η διεργασία που τυπώνει το αποτέλεσμα (C) κάνει signal 2 φορές την A και μία φορά την B, ενώ κάθε μία από αυτές τις διεργασίες πριν τερματίσουν κάνουν μία φορά signal την C. Η C από την πλευρά της κάνει 3 wait πριν ξεκινήσει. Συνεπώς για να γενικευτεί αυτό το σχήμα για την περίπτωση όπου η διεργασία B εκτελούσε n = n – K χρειάζεται απλά η C να κάνει signal K φορές την A και μία φορά την B, ενώ κάθε μία από αυτές πριν τερματίσουν κάνουν μία φορά signal την C. Η C από την πλευρά της να κάνει K + 1 wait πριν ξεκινήσει.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <math.h>

#include "pipesem.h"
#include "proc-common.h"

// This is a pointer to a shared memory area.
// It holds an integer value, that is manipulated
// concurrently by all children processes.
int *shared_memory;
// semaphores
struct pipesem *sems[3];

// functions
void proc_A(void);
void proc_B(void);
void proc_C(void);

// Use a NULL-terminated array of pointers to functions.
// Each child process gets to call a different pointer.
typedef void proc_fn_t(void);
static proc_fn_t *proc_funcs[]={proc_A,proc_B,proc_C,NULL};

int main(int argc,char *argv[])
{
    int i,status;
    pid_t pid;
    proc_fn_t *proc_fn;
    // initialize a semaphore for every process
    for(i=0;i<3;i++) sems[i]=(struct pipesem*)malloc(sizeof(struct pipesem));
    for(i=0;i<3;i++) pipesem_init(sems[i],0);
    // create a shared memory area
    shared_memory=create_shared_memory_area(sizeof(int));
    *shared_memory=1;
    // signal the print process to start the cycle
    for(i=0;i<3;i++) pipesem_signal(sems[2]);
    // fork all three processes
    for(i=0;(proc_fn=proc_funcs[i])!=NULL;i++)
    {
        printf("%lu fork()\n",(unsigned long)getpid());
        pid=fork();
        if(pid<0)
        {
            perror("parent: fork");
            exit(1);
        }
        if(pid) continue;
        // child process
        proc_fn();
        exit(1);
    }
    // parent waits for all children to terminate
    for(i=0;i<3;i++) wait(&status);
    // destroy the semaphores
```

```
    for(i=0;i<3;i++) pipesem_destroy(sems[i]);
    return 0;
}

// Proc A: n = n + 1
void proc_A(void)
{
    volatile int *n=&shared_memory[0];
    for(;;)
    {
        pipesem_wait(sems[0]);
        *n=*n+1;
        pipesem_signal(sems[2]);
    }
    exit(1);
}

// Proc B: n = n - 2
void proc_B(void)
{
    volatile int *n=&shared_memory[0];
    for(;;)
    {
        pipesem_wait(sems[1]);
        *n=*n-2;
        pipesem_signal(sems[2]);
    }
    exit(1);
}

// Proc C: print n
void proc_C(void)
{
    int val,i;
    volatile int *n=&shared_memory[0];
    for(;;)
    {
        for(i=0;i<3;i++) pipesem_wait(sems[2]);
        val=*n;
        printf("Proc C: n = %d\n",val);
        if(val!=1) printf("\t...Aaaaargh!\n");
        for(i=0;i<2;i++) pipesem_signal(sems[0]);
        pipesem_signal(sems[1]);
    }
    exit(1);
}
```

Έξοδος εκτέλεσης προγράμματος:

Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1
Proc C: n = 1 …

3 Προαιρετικές ερωτήσεις:

1. Για να υλοποιήσουμε σημαφόρους με χρήση παρακολουθητών θα τοποθετούσαμε το κρίσιμο τμήμα μέσα στον παρακολουθητή και θα είχαμε δύο μεθόδους οι οποίες θα εκτελούνταν πάνω στο pipe, τις:
   wait() : Όπου η διεργασία αδρανοποιείται μέχρι να κληθεί η signal().
   signal() : Όπου ενεργοποιεί ακριβώς μία από τις αδρανοποιημένες διεργασίες.

2. Το πρόγραμμα rand-fork θα θέλαμε να τυπώνει 10 τυχαίους αριθμούς. Αντ' αυτού τυπώνει 10 ίδιους αριθμούς. Αυτό συμβαίνει γιατί όλες οι διεργασίες αρχικοποιούν την ψευδοτυχαία γεννήτρια με το ίδιο seed, συνεπώς η rand() την ίδια ακολουθία αποτελεσμάτων. Όμως κάθε διεργασία καλεί την rand() μία μόνο φορά και έτσι και οι 10 καταλήγουν να τυπώσουν 10 φορές τον ίδιο αριθμό που είναι ο πρώτος που δίνει η rand() για την αρχικοποίηση με το συγκεκριμένο seed.