ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Σχολη ΗΜ&ΜΥ
Λειτουργικά Συστήματα
Άσκηση 3
Ακ. Έτος 2012-13

Παπαδιάς Σεραφείμ | ΑΜ: 03109193
Παπαδημητρίου Κων/νος | ΑΜ: 03108769
Ημερομηνία: 31/01/2013

Άσκηση 1.1 Υλοποίηση σημαφόρων με σωληνώσεις του UNIX

Ο κώδικας που γράψαμε, καθώς και η έξοδος του προγράμματος, βρίσκονται στα παραρτήματα, στο τέλος αυτής της αναφοράς.

Έρωτησεις

1. *Τι ακριβώς κάνει το pipesem-test.c; Πόσες διεργασίες τρέχουν και πώς αυτές συγχρονίζονται;*

Το pipesem-test.c δημιουργεί μια διεργασία-παιδί και στη συνέχεια κάνει wait στο σημαφόρο. Το παιδί "κοιμάται" για 5 δευτερόλεπτα και στη συνέχεια κάνει signal το σημαφόρο και τερματίζει. Ο πατέρας ξεμπλοκάρει και τερματίζει κι αυτός (τυπώνοντας και τα ανάλογα μηνύματα κάθε φορά).

2. *Ποια είναι η βασική αρχή λειτουργίας της βιβλιοθήκης pipesem.c; Πώς εξασφαλίζεται ότι η πράξη wait σε σημαφόρο μηδενικής ή αρνητικής τιμής μπλοκάρει; Τι σημαίνει αυτό για την υφιστάμενη σωλήνωση του UNIX;*

Η βασική αρχή λειτουργίας της βιβλιοθήκης pipesem.c είναι ότι όταν μια διεργασία κάνει read στο άκρο ανάγνωσης του pipe, μπλοκάρει μέχρι να βρεθούν δεδομένα για ανάγνωση. Έτσι όταν μια διεργασία εισέρχεται στο critical section, πρώτα θα κάνει read ένα token ώστε να μπλοκάρει έως ότου βρεθούν δεδομένα στο pipe, και όταν εξέρχεται θα κάνει write ένα token ώστε η επόμενη διεργασία να μπορέσει να μπει στο critical section με τη σειρά της.

3. *Θα μπορούσε αυτός ο τρόπος υλοποίησης να οδηγήσει σε λιμοκτονία (starvation); Πώς επηρεάζεται η λειτουργία του από τον τρόπο με τον οποίο το ΛΣ υλοποιεί την ανάγνωση από pipes, όταν πολλές διεργασίες χρειάζεται να περιμένουν σε άδεια σωλήνωση;*

Θα μπορούσε, αφού αυτό εξαρτάται από το πως το ΛΣ διαχειρίζεται τις σωληνώσεις. Το UNIX χρησιμοποιεί λίστες FIFO για τη διαχείρηση των αιτήσεων ανάγνωσης στο άκρο ανάγνωσης, έτσι είμαστε σίγουροι πως δε θα υπάρξει λιμοκτονία.

Άσκηση 1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Ο κώδικας που γράψαμε, καθώς και η έξοδος του προγράμματος, βρίσκονται στα παραρτήματα, στο τέλος αυτής της αναφοράς.

Έρωτησεις

1. *Πόσοι σημαφόροι χρειάζονται για το σχήμα συγχρονισμού που υλοποιείτε;*

Χρειάζονται τόσοι σημαφόροι όσες και οι διεργασίες που έχουν αναλάβει ένα μέρος του υπολογισμού. Για την ακρίβεια, ο υπολογισμός μίας γραμμής μπορεί να γίνει τελείως ανεξάρτητα από κάθε διεργασία. Το πρόβλημα εμφανίζεται στην εκτύπωση της κάθε γραμμής. Έτσι για να εκτυπώσει μια διεργασία κάνει πρώτα wait στον αντίστοιχο σημαφόρο και όταν εκτυπώσει κάνει signal το σημαφόρο της διεργασίας που έχει αναλάβει

την επόμενη γραμμή.

2. *Πόσος χρόνος απαιτείται για την ολοκλήρωση του σειριακού και του παράλληλου προγράμματος με δύο διεργασίες υπολογισμού;*

Μετά από μετρήσεις που κάναμε με τη χρήση της time σε μηχάνημα με επεξεργαστή 2 πυρήνων, βρήκαμε πως το παράλληλο πρόγραμμα απαιτεί 0.475s, ενώ το σειριακό απαιτεί 0.785s.

3. *Το παράλληλο πρόγραμμα που φτιάξατε, εμφανίζει επιτάχυνση; αν όχι, γιατί; τι πρόβλημα υπάρχει στο σχήμα συγχρονισμού που έχετε υλοποιήσει;*

Εμφανίζεται επιτάχυνση περίπου 39.5% όταν ο υπολογισμός γίνεται παράλληλα. Το κρίσιμο τμήμα αποτελείται μόνο από τη φάση της εξόδου και μια διεργασία μπορεί να υπολογίζει μια γραμμή ανεξάρτητα από τη κατάσταση μιας άλλης διεργασίας.

4. *Τι συμβαίνει στο τερματικό αν πατήσετε Ctrl-C ενώ το πρόγραμμα εκτελείται; σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμάτων; πώς θα μπορούσατε να επεκτείνεται το mandel.c σας ώστε να εξασφαλίσετε ότι ακόμη κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του;*

Οι γραμμές θα έχουν αλλάξει χρώμα και θα έχουν το χρώμα που είχε ο τελευταίος χαρακτήρας που τυπώθηκε. Για να αντιμετωπίσουμε κάτι τέτοιο, πρέπει να επεκτείνουμε το mandel.c με ένα signal handler, ώστε όταν έρθει kill ή interrupt signal να εκτελεστεί η reset_xterm_color πριν τερματίσει το πρόγραμμα.

Άσκηση 1.3 Ταυτόχρονη πρόσβαση σε μοιραζόμενους πόρους

Ο κώδικας που γράψαμε, καθώς και η έξοδος του προγράμματος, βρίσκονται στα παραρτήματα, στο τέλος αυτής της αναφοράς.

Έρωτησεις

1. *Ποιος είναι ο σκοπός της κλήσης create_shared_memory_area(sizeof(int)), πριν από τη δημιουργία των τριών διεργασιών;*

Σκοπός της create_shared_memory_area(sizeof(int)) είναι να δημιουργηθεί ένας χώρος στην μνήμη στον οποίο θα έχουν πρόσβαση οι 3 διεργασίες-παιδιά αργότερα.

2. *Πώς θα μπορούσε να γενικευτεί το σχήμα συγχρονισμού που υλοποιήσατε για την περίπτωση στην οποία η* P_B *εκτελούσε n = n-K, με K δεδομένη θετική σταθερά;*

Η διεργασία P_A θα πρέπει να κάνει K φορές signal το σημαφόρο της και στην επόμενη επανάληψη να κάνει signal το σημαφόρο της διεργασίας P_B. Έτσι έχουμε μια μεταβλητή i (αρχικοποιημένη σε 1) την οποία αυξάνουμε σε κάθε επανάληψη και αν i < K κάνουμε signal το σημαφόρο της P_A, αλλιώς κάνουμε i=1 και signal το σημαφόρο της P_B. *(Σημείωση: ο ενημερωμένος κώδικας της* P_A *δίνεται στο παράρτημα!)*

ΠΑΡΑΡΤΗΜΑ 1

1. Κώδικας που γράψαμε για την άσκηση 1

```
// file: pipesem.h

#ifndef PIPESEM_H__
#define PIPESEM_H__

struct pipesem {
        /*
         * Two file descriptors:
         * one for the read and one for the write end of a pipe
         */
        int rfd;
        int wfd;
};

/*
 * Function prototypes
 */
void pipesem_init(struct pipesem *sem, int val);
void pipesem_wait(struct pipesem *sem);
void pipesem_signal(struct pipesem *sem);
void pipesem_destroy(struct pipesem *sem);

#endif /* PIPESEM_H__ */
```

==========

```
// file: pipesem.c

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#include "pipesem.h"

void
pipesem_init(struct pipesem *sem, int val)
{
   /* δεν χρειάζεται αρχικοποίηση του token γιατί
    * δεν μας νοιάζει το τί θα γράψουμε (ή θα διαβάσουμε
    * στην pipesem_wait()) από το pipe αλλά το πόσο.
    * */
   int token;
   if (pipe(&sem->rfd) < 0)
   {
        perror("init: pipe");
        exit(EXIT_FAILURE);
   }

   while (val>0)
   {
        if (write(sem->wfd, &token, sizeof(int)) == -1){
           perror("init: write");
```

```
            exit(EXIT_FAILURE);
        }
        val--;
    }
}

void
pipesem_wait(struct pipesem *sem)
{
    int token;
    if (read(sem->rfd, &token, sizeof(int)) == -1){
        perror("wait: read");
        exit(EXIT_FAILURE);
    }
}

void
pipesem_signal(struct pipesem *sem)
{
    int token;
    if (write(sem->wfd, &token, sizeof(int)) == -1){
        perror("signal: write");
        exit(EXIT_FAILURE);
    }
}

void
pipesem_destroy(struct pipesem *sem)
{
    close(sem->rfd);
    close(sem->wfd);
}
```

2. Κώδικας που γράψαμε για την άσκηση 2

Σημείωση: Είναι μόνο ο κώδικας του mandel.c καθώς μόνο αυτό το αρχείο αλλάξαμε.

```
/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */

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

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

#define MANDEL_MAX_ITERATION 100000
#define NCHILDREN 8
```

```
/**************************
 * Compile-time parameters *
 **************************/

/*
 * Output at the terminal is 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;

/*
 * 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;
        int 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 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("compute_and_output_mandel_line: write point");
			exit(1);
		}
	}

	/* Now that the line is done, output a newline character */
	if (write(fd, &newline, 1) != 1) {
		perror("compute_and_output_mandel_line: write newline");
		exit(1);
	}
}

void
proc_task(int fd, int proc, struct pipesem *cur_sem, struct pipesem *next_sem){
   int line;
   int color_val[x_chars];
   for (line = proc; line < y_chars; line += NCHILDREN){
	compute_mandel_line(line, color_val);

	// critical_section
	pipesem_wait(cur_sem);
	output_mandel_line(fd, color_val);
	pipesem_signal(next_sem);
   }
}

int main(void)
{
	xstep = (xmax - xmin) / x_chars;
	ystep = (ymax - ymin) / y_chars;

	/*
	 * draw the Mandelbrot Set, one line at a time.
	 * Output is sent to file descriptor '1', i.e., standard output.
	 */

	int i;
	pid_t pid;

	/* define and initialize all semaphores
	 * sem[0] takes a 'cookie' */
	struct pipesem sem[NCHILDREN];
	for (i=0; i<NCHILDREN; i++)
	   pipesem_init(&sem[i], 0);
```

```
        pipesem_signal(&(sem[0]));

        for (i=0; i<NCHILDREN; i++){
           if ((pid=fork()) == -1){
                perror("main: fork");
                exit(EXIT_FAILURE);
           }
           if (pid == 0){
                proc_task(1, i, &sem[i], &sem[(i+1)%NCHILDREN]);
                exit(EXIT_SUCCESS);
           }
        }

        int status;
        for (i=0; i<NCHILDREN; i++)
           wait(&status);
        reset_xterm_color(1);
        return 0;
}
```

3. Κώδικας που γράψαμε για την άσκηση 3

Σημείωση: Όπως και παραπάνω αντιγράφουμε μόνο τον κώδικα που αλλάξαμε.

```
//file: procs-shm.c

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"
#include "pipesem.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;
/* ... */

/* Proc A: n = n + 1 */
void proc_A(struct pipesem *sem)
{
        volatile int *n = &shared_memory[0];
        /* ... */
        int i=0;
        for (;;) {
                /* ... */
                pipesem_wait(&sem[0]);
                *n = *n + 1;
                /* ... */
```

```
                i = i%2;
                pipesem_signal(&sem[i]);
                i++;
        }

        exit(0);
}

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

        exit(0);
}

/* Proc C: print n */
void proc_C(struct pipesem *sem)
{
        int val;

        volatile int *n = &shared_memory[0];
        /* ... */

        for (;;) {
                /* ... */
                pipesem_wait(&sem[2]);
                val = *n;
                printf("Proc C: n = %d\n", val);
                if (val != 1) {
                        printf("     ...Aaaaaargh!\n");
                }
                /* ... */
                pipesem_signal(&sem[0]);
        }
        exit(0);
}

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

int main(void)
{
        int i;
        int status;
```

```
        pid_t p;
        proc_fn_t *proc_fn;

        /* ... */
        struct pipesem sem[3];
        for (i=0; i<3; i++)
           pipesem_init(&sem[i], 0);
        pipesem_signal(&sem[0]);

        /* Create a shared memory area */
        shared_memory = create_shared_memory_area(sizeof(int));
        *shared_memory = 1;

        for(i = 0; (proc_fn = proc_funcs[i]) != NULL; i++) {
                printf("%lu fork()\n", (unsigned long)getpid());
                p = fork();
                if (p < 0) {
                        perror("parent: fork");
                        exit(1);
                }
                if (p != 0) {
                        /* Father */
                        continue;
                }
                /* Child */
                proc_fn(sem);
                assert(0);
        }

        /* Parent waits for all children to terminate */
        for (; i >0; i--)
                wait(&status);

        return 0;
}
```

Κώδικας της P_A ***που γράψαμε για την γενικευμένη άσκηση 3***

```
...
#define K 10
/* Proc A: n = n + 1 */
void proc_A(struct pipesem *sem)
{
        volatile int *n = &shared_memory[0];
        /* ... */
        int i=1;
        for (;;) {
                /* ... */
                pipesem_wait(&sem[0]);
                *n = *n + 1;
                /* ... */
                if (i<K)
                {
                   pipesem_signal(&sem[0]);
                   i++;
                }
                else
                {
```

```
                    i=1;
                    pipesem_signal(&sem[1]);
                }
        }

        exit(0);
}
...
```

ΠΑΡΑΡΤΗΜΑ 2

1. Αποτελέσματα από την άσκηση 1

```
$ ./pipesem-test
Parent: waiting on semaphore
Child: sleeping for two seconds
Child: signaling semaphore
Parent: signaled, program should terminate.
```

2. Αποτελέσματα από την άσκηση 2

Σημείωση: Το αποτέλεσμα της άσκησης 2 είναι ένα ωραίο fractal στο terminal το οποίο δυστυχώς για ευνόητους λόγους δεν μπορούμε να αντιγράψουμε εδώ!

3. Αποτελέσματα από την άσκηση 3

```
$ ./procs-shm
19307 fork()
19307 fork()
19307 fork()
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
...and so on
```

Υποσημείωση: Έχουμε επισυνάψει τα αρχεία όλου του κώδικα της 3ης σειράς.