Λειτουργικά Συστήματα

Ομάδα D10

Διονύσης Ζήνδρος
Πέτρος Αγγελάτος

Άσκηση 3

Ερωτήσεις

1.1 Ερωτήσεις

1. Δημιουργεί δύο διεργασίες από τις οποίες το παιδί στέλνει ένα σήμα στον πατέρα μέσω ενός σημαφόρου.
2. Η υλοποίηση των σημαφόρων βασίζεται στο γεγονός ότι η κλήση συστήματος read μπλοκάρει έως ότου να υπάρχουν δεδομένα να διαβαστούν από το σωλήνα. Αυτά γίνονται διαθέσιμα όταν θέλουμε να ξεμπλοκάρουμε τη διεργασία που έκανε read, δηλαδή όταν έρθει κάποιο σήμα στο σημαφόρο.
3. Αν πολλές διεργασίες κάνουν read ταυτόχρονα σε ένα σημαφόρο που αντιστοιχεί σε ένα resource που δεν είναι διαθέσιμο, το λειτουργικό σύστημα θα πρέπει να φροντίσει να διαβάσουν με σειρά προτεραιότητας για να αποφευχθεί η λιμοκτονία. Διαφορετικά μπορεί μία διεργασία να περιμένει να κάνει read και μια άλλη που θα επιχειρήσει να κάνει read αργότερα να της πάρει την προτεραιότητα με αποτέλεσμα η αρχική να μην ενεργοποιηθεί ποτέ.

1.2 Ερωτήσεις

Σε αυτή την άσκηση αποφασίσαμε να κάνουμε τους υπολογισμούς στα παιδιά της διεργασίες παράλληλα, ενώ ο πατέρας να τυπώνει το σύνολο Mandelbrot καθώς αυτό υπολογίζεται. Έτσι, ο υπολογισμός παραλληλοποιείται, ενώ η εμφάνιση είναι σειριακή. Η επικοινωνία ανάμεσα στον πατέρα και στα παιδιά γίνεται μέσω ενός σωλήνα για κάθε παιδί, ενώ χρησιμοποιούνται σημαφόροι για να ειδοποιηθεί ο πατέρας για την ολοκλήρωση του υπολογισμού μίας ολόκληρης γραμμής.

1. Το πλήθος των σημαφόρων είναι ίσο με το πλήθος των παιδιών
2. Σε διπύρινο επεξεργαστή, ο χρόνος που χρειάζεται το σειριακό πρόγραμμα είναι λίγο λιγότερος από το διπλάσιο περίπου του χρόνου που κάνει το παράλληλο πρόγραμμα.
3. Ναι, εμφανίζει.
4. Το τερματικό εμφανίζει τα γράμματα στο τελευταίο χρώμα που έμεινε ενεργό κατά τη ζωγραφική του συνόλου. Θα μπορούσε να αποφευχθεί κάτι τέτοιο πιάνοντας το SIGINT που προκαλεί το Ctrl-C του χρήστη ώστε το πρόγραμμα να μην πεθάνει αμέσως. Στη συνέχεια θα μπορούσαμε να επαναφέρουμε το χρώμα σε λευκό και να τερματίσουμε το πρόγραμμα πρόωρα.

1.3 Ερωτήσεις

1. Η συγκεκριμένη κλήση δημιουργεί ένα τμήμα μνήμης που είναι κοινό ανάμεσα σε όλες τις διεργασίες. Χρησιμοποιείται έτσι ώστε οι τιμές που γράφει στη μεταβλητή η μία διεργασία να διαβάζονται μετά από την άλλη.
2. Θα επεκτείναμε τη βιβλιοθήκη σημαφόρων έτσι ώστε να υποστηρίζει wait και signal με παράμετρο το πλήθος που επιθυμούμε. Αυτό θα μπορούσε να υλοποιηθεί και πάλι με σωλήνες του UNIX.

Πηγαίος Κώδικας

pipesem.c

```
/*
 * pipesem.c
 */

#include <unistd.h>
#include <stdio.h>
#include "pipesem.h"

void pipesem_init(struct pipesem *sem, int val)
{
   int i;
   int f[ 2 ];
   int status;

   status = pipe(f);
   if (status < 0) {
      perror("Could not create semaphore");
      return;
   }

   sem->rfd = f[ 0 ];
   sem->wfd = f[ 1 ];

   for (i = 0; i < val; ++i) {
      pipesem_signal(sem);
   }
}

void pipesem_wait(struct pipesem *sem)
{
   char buffer[ 1 ];
   int status;

   status = read(sem->rfd, buffer, 1);
```

```
    if (status < 0) {
        perror("Could not wait on semaphore");
        return;
    }
}

void pipesem_signal(struct pipesem *sem)
{
    int status;
    status = write(sem->wfd, "_", 1);

    if (status < 0) {
        perror("Could not signal semaphore");
        return;
    }
}

void pipesem_destroy(struct pipesem *sem)
{
    int status;

    status = close(sem->rfd);
    if (status < 0) {
        perror("Could not destroy semaphore (read end indestructible)");
        return;
    }

    status = close(sem->wfd);
    if (status < 0) {
        perror("Could not destroy semaphore (write end indestructible)");
        return;
    }
}
```

mandel.c

```
/*
 * 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 "mandel-lib.h"
#include "pipesem.h"

#define MANDEL_MAX_ITERATION 100000
#define NCHILDREN 1

/***************************
 * 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 compute_and_output_mandel_line(int fd, int line)
{
    /*
```

```
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];

    compute_mandel_line(line, color_val);
    output_mandel_line(fd, color_val);
}

int main(void)
{
    int line;
    int buffer[x_chars];
    struct pipesem sems[NCHILDREN];
    int pipes[NCHILDREN][2];

    xstep = (xmax - xmin) / x_chars;
    ystep = (ymax - ymin) / y_chars;

    int pids[NCHILDREN];
    int i,j;
    for (i = 0; i < NCHILDREN; i++) {
        pipe(pipes[i]);
        pipesem_init(&sems[i], 0);
        pids[i] = fork();
        if (pids[i] < 0) {
            perror("Failed to fork");
            return 0;
        }
        else if (pids[i] == 0) {
            close(pipes[i][0]);

            for (j = i; j < y_chars; j += NCHILDREN) {
                compute_mandel_line(j, buffer);
                int status;
                int bytes_written = 0;
                //printf( "Process %d. Computing line %d\n", i, j );
                while (bytes_written < x_chars * sizeof(int)) {
                    status = write(pipes[i][1], (void*) buffer + bytes_written,
x_chars * sizeof(int) - bytes_written);
                    if (status < 0) {
                        perror("Could not write to pipe");
                        //TODO Inform your mama
                        return 0;
                    }
                    bytes_written += status;
                }
                pipesem_signal(&sems[i]);
```

```
            }
            close(pipes[i][1]);
            return 0;
        }
        else {
            close(pipes[i][1]);
        }
    }

    /*
     * draw the Mandelbrot Set, one line at a time.
     * Output is sent to file descriptor '1', i.e., standard output.
     */
    int status;
    int bytes_read;
    for (line = 0; line < y_chars; line++) {
        bytes_read = 0;
        pipesem_wait(&sems[line % NCHILDREN]);
        while (bytes_read < x_chars * sizeof(int)) {
            //printf( "Reading %d bytes\n", x_chars * sizeof(int) - bytes_read);
            status = read(pipes[line % NCHILDREN][0], (void*) buffer +
bytes_read, x_chars * sizeof(int) - bytes_read);
            if (status < 0) {
                perror("Could not read from pipe");
                return 0;
            }
            bytes_read += status;
        }
        output_mandel_line(1, buffer);
    }

    reset_xterm_color(1);
    return 0;
}
```

ask-3-3.c

```
/*
 * procs-shm.c
 *
 * A program to create three processes,
 * working with a shared memory area.
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Operating Systems
 *
 */
```

```
#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;
struct pipesem sem[ 3 ];

/* Proc A: n = n + 1 */
void proc_A(void)
{
        volatile int *n = &shared_memory[0];

        for (;;) {
      pipesem_wait(&sem[2]);
                *n = *n + 1;
      pipesem_signal(&sem[0]);
        }

        exit(0);
}

/* Proc B: n = n - 2 */
void proc_B(void)
{
        volatile int *n = &shared_memory[0];

        for (;;) {
      pipesem_wait(&sem[0]);
      pipesem_wait(&sem[0]);
                *n = *n - 2;
      pipesem_signal(&sem[1]);
        }
```

```
        exit(0);
}

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

        volatile int *n = &shared_memory[0];

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

   pipesem_init(&sem[0], 0);
   pipesem_init(&sem[1], 1);
   pipesem_init(&sem[2], 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();
                assert(0);
        }

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

        return 0;
}
```

Output προγραμμάτων

```
dionyziz@europa ~/ntua/os/ex3/sync (master) % ./pipesem-test
Parent: waiting on semaphore
Parent: signaled, program should terminate.
Child: sleeping for five seconds
Child: signaling semaphore
dionyziz@europa ~/ntua/os/ex3/sync (master) % ./mandel
(nifty mandelbrot set)
dionyziz@europa ~/ntua/os/ex3/sync (master) % ./ask3-3
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
```