



ΧΑΡΟΚΟΠΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΜΑΤΙΚΗΣ

‘Αρβι Χότζα

2^η Εργασία στο μάθημα **Λειτουργικά Συστήματα**

Ταύρος, 04 Σεπτεμβρίου 2024

Περιεχόμενα

Άσκηση 2	3
Κώδικας	3
Τρόπος Εκτέλεσης	7
Ενδεικτικές εκτελέσεις (screenshots):	7
Βασική διεργασία η οποία ελέγχει τη δημιουργία των κατάλληλων διεργασιών και τον έλεγχο του προγράμματος	7
Screenshots	7
Συγχρονισμός των 2 διεργασιών εγγραφής ανάγνωσης	8
Screenshots	8
Διαχείριση σημάτων	8
Screenshots	8
Δημιουργία νημάτων και πέρασμα παραμέτρων	10
Screenshots	10
Συγχρονισμός νημάτων και σωστή διαδικασία μέτρησης αποτελέσματος	10
Screenshots	10
Ομαλή εκτέλεση προγράμματος, error handling, τεκμηρίωση	10
Screenshots	11
Ερωτήσεις εκφώνησης	11
Προσοχή στο συγχρονισμό μέσω σημαφόρων των 2 διεργασιών. Πού πρέπει να δηλώνεται ο σημαφόρος για να είναι προσβάσιμος και γνωστός και στις 2 διεργασίες; Σε τι τιμή θα πρέπει να αρχικοποιείται και τι πράξεις να κάνει πάνω του κάθε διεργασία;	11
Χρειάζεται να διαμοιράζονται αυτές οι δύο διεργασίες κάποιον file descriptor;	11
Χρειάζεται να μοιράζονται όλα τα νήματα της 2ης διεργασίας-αναγνώστη κάποιον file descriptor ανάγνωσης; Χρειάζεται προστασία αυτός από ταυτόχρονη πρόσβαση;	12
Προσοχή στην ταυτόχρονη πρόσβαση στον κεντρικό πίνακα της μέτρησης. Είναι κρίσιμη περιοχή; Χρειάζεται προστασία; Δικαιολογήστε την απάντησή σας	12
Γενικά Σχόλια/Παρατηρήσεις	12
Με δυσκόλεψε / δεν υλοποίησα	12
Συνοπτικός Πίνακας	13

Άσκηση 2

Κώδικας

Ο κώδικας της 2ης εργασίας που δημιουργήθηκε μαζί με τα σχόλια είναι:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <fcntl.h>
#include <time.h>
#include <string.h>

#define NUM_THREADS 4
#define LINES 100
#define NUMBERS_PER_LINE 50

sem_t *sem;
int total_sum = 0;
pthread_mutex_t sum_mutex;

void signal_handler(int sig) {
    char response;
    printf("\nReceived signal %d. Do you want to terminate the program? (y/n):", sig);
    scanf(" %c", &response);
    if (response == 'y' || response == 'Y') {
        // Απελευθέρωση των πόρων
        if (sem_close(sem) != 0) {
```

```

        perror("Failed to close semaphore");
        exit(1);
    }
    if (sem_unlink("/file_sem") != 0) {
        perror("Failed to unlink semaphore");
        exit(1);
    }
    if (pthread_mutex_destroy(&sum_mutex) != 0) {
        perror("Failed to destroy mutex");
        exit(1);
    }
    exit(sig);
}
}

void process1() {
    FILE *file = fopen("data.txt", "w");
    if (file == NULL) {
        perror("Failed to open file");
        exit(1);
    }

    srand(time(NULL)); // Για τη δημιουργία τυχαίων αριθμών
    for (int i = 0; i < LINES; i++) {
        for (int j = 0; j < NUMBERS_PER_LINE; j++) {
            fprintf(file, "%d ", rand() % 101);
        }
        fprintf(file, "\n");
    }

    fclose(file);
    printf("File written successfully by Process 1\n");

    if (sem_post(sem) == -1) { // Απελευθερώνει το σηματοφόρο για να ξεκινήσει η
Process 2
        perror("sem_post failed");
        exit(1);
    }
}

void* thread_function(void *arg) {
    FILE *file = fopen("data.txt", "r");

```

```

    if (file == NULL) {
        perror("Failed to open file");
        exit(1);
    }

    int thread_id = *(int*)arg;
    int sum = 0, number;
    int line_count = 0;
    char line[1024];

    // Κάθε νήμα διαβάζει τη γραμμή που του αντιστοιχεί
    for (int i = 0; i < LINES; i++) {
        if (i % NUM_THREADS == thread_id) {
            fgets(line, sizeof(line), file);
            line_count++;
            char *token = strtok(line, " ");
            while (token != NULL) {
                number = atoi(token);
                sum += number;
                token = strtok(NULL, " ");
            }
        }
    }

    pthread_mutex_lock(&sum_mutex);
    total_sum += sum;
    pthread_mutex_unlock(&sum_mutex);

    printf("Thread %d processed %d lines. Local sum: %d\n", thread_id,
line_count, sum);
    fclose(file);
    return NULL;
}

void process2() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    if (sem_wait(sem) == -1) { // Περιμένει να τελειώσει η Process 1
        perror("sem_wait failed");
        exit(1);
    }
}

```

```

// Δημιουργία των νημάτων
for (int i = 0; i < NUM_THREADS; i++) {
    thread_ids[i] = i;
    pthread_create(&threads[i], NULL, thread_function,
(void*)&thread_ids[i]);
}

// Αναμονή για όλα τα νήματα
for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
}

printf("Total sum of all numbers: %d\n", total_sum);
}

int main() {
    signal(SIGINT, signal_handler);
    signal(SIGTERM, signal_handler);
    // Δημιουργία σημαφόρου
    sem = sem_open("/file_sem", O_CREAT, 0644, 0);
    if (sem == SEM_FAILED) {
        perror("Failed to create semaphore");
        exit(1);
    }
    if (pthread_mutex_init(&sum_mutex, NULL) != 0) {
        perror("Failed to initialize mutex");
        exit(1);
    }
    pid_t pid = fork();
    if (pid < 0) {
        perror("Failed to fork");
        exit(1);
    } else if (pid == 0) {
        process1();
        exit(1);
    } else {
        process2();
        if (waitpid(pid, NULL, 0) < 0) {
            perror("Failed to wait for child process");
            exit(1);
        }
    }
}

```

```
}  
if (sem_close(sem) != 0) {  
    perror("Failed to close semaphore");  
    exit(1);  
}  
if (sem_unlink("/file_sem") != 0) {  
    perror("Failed to unlink semaphore!");  
    exit(1);  
}  
if (pthread_mutex_destroy(&sum_mutex) != 0) {  
    perror("Failed to destroy mutex");  
    exit(1);  
}  
return 0;  
}
```

Τρόπος Εκτέλεσης

gcc it22121.c

Με αυτόν τον τρόπο δημιουργείται το εκτελέσιμο (a.out)

Ενδεικτικές εκτελέσεις (screenshots):

Βασική διεργασία η οποία ελέγχει τη δημιουργία των κατάλληλων διεργασιών και τον έλεγχο του προγράμματος

Screenshots

```
int main() {
    signal(SIGINT, signal_handler);
    signal(SIGTERM, signal_handler);
    // Δημιουργία σηματοφύρου
    sem = sem_open("/file_sem", 0_CREAT, 0644, 0);
    if (sem == SEM_FAILED) {
        perror("Failed to create semaphore");
        exit(1);
    }
    if (pthread_mutex_init(&sum_mutex, NULL) != 0) {
        perror("Failed to initialize mutex");
        exit(1);
    }
    pid_t pid = fork();
    if (pid < 0) {
        perror("Failed to fork");
        exit(1);
    } else if (pid == 0) {
        process1();
        exit(1);
    } else {
        process2();
        if (waitpid(pid, NULL, 0) < 0) {
            perror("Failed to wait for child process");
            exit(1);
        }
    }
    if (sem_close(sem) != 0) {
        perror("Failed to close semaphore");
        exit(1);
    }
    if (sem_unlink("/file_sem") != 0) {
        perror("Failed to unlink semaphore!");
        exit(1);
    }
    if (pthread_mutex_destroy(&sum_mutex) != 0) {
        perror("Failed to destroy mutex");
        exit(1);
    }
    return 0;
}
```


Στην βασική διεργασία, αρχικά δημιουργείται ένας σημαφόρος και ένας mutex για συγχρονισμό. Στη συνέχεια, γίνεται κλήση στη `fork()` για τη δημιουργία μιας νέας διεργασίας. Αν η `fork()` αποτύχει, το πρόγραμμα τερματίζεται με μήνυμα σφάλματος. Η διεργασία παιδί εκτελεί τη συνάρτηση `process1()`, ενώ η διεργασία γονέας εκτελεί τη συνάρτηση `process2()` και περιμένει την ολοκλήρωση της διεργασίας παιδιού με τη χρήση της `waitpid()`. Τέλος, ο σημαφόρος κλείνει και αποσυνδέεται, και ο mutex καταστρέφεται πριν το πρόγραμμα τερματιστεί.

Συγχρονισμός των 2 διεργασιών εγγραφής ανάγνωσης

Ο Συγχρονισμός των 2 διεργασιών γίνεται με την χρήση του σημαφόρου και του mutex.

Ο σημαφόρος χρησιμοποιείται για να εξασφαλιστεί ότι θα δημιουργηθεί πρώτα το αρχείο `data.txt` στην πρώτη διεργασία και έπειτα θα συνεχίσει την εκτέλεση η δεύτερη διεργασία. Συγκεκριμένα, στην `main()` ορίζεται με `sem = sem_open("/file_sem", O_CREAT, 0644, 0);`, όπου `"/file_sem"` το όνομα του σημαφόρου, `O_CREAT` αν δεν υπάρχει να δημιουργηθεί, `0644` τα δικαιώματα και `0` η αρχική τιμή του σημαφόρου. Η αρχική τιμή `0` σημαίνει ότι οποιαδήποτε διεργασία που θα καλέσει τη `sem_wait` θα μπλοκαριστεί μέχρι κάποια άλλη διεργασία να καλέσει τη `sem_post` και να αυξήσει την τιμή του σημαφόρου.

Ο mutex χρησιμοποιείται για να εξασφαλιστεί ότι δεν θα υπάρχει ταυτόχρονη πρόσβαση από τα νήματα στην κρίσιμη περιοχή. Συγκεκριμένα, στην `main()` αρχικοποιείται ο mutex, εφόσον έχει οριστεί προηγουμένως, και χρησιμοποιείται για να προστατεύσει την μεταβλητή `total_sum`. Αυτό επιτυγχάνεται κλειδώνοντας τον mutex πριν την πράξη `total_sum += sum;` και ξεκλειδώνοντας τον μετά.

Screenshots

```
pthread_mutex_lock(&sum_mutex);
total_sum += sum;
pthread_mutex_unlock(&sum_mutex);
```

Διαχείριση σημάτων

Για την διαχείριση σημάτων χρησιμοποίησα την βιβλιοθήκη `signal.h` και εγκατέστησα τους χειριστές σημάτων στην αρχή της `main()` για τα δύο σήματα. Αφότου λάβει κάποιο από τα δύο σήματα `SIGINT` ή `SIGTERM` ρωτάει τον χρήστη εάν είναι σίγουρος για αυτήν την ενέργεια και έπειτα απελευθερώνει τους πόρους που δεσμεύει και τερματίζει το πρόγραμμα.

Screenshots

```

void signal_handler(int sig) {
    char response;
    printf("\nReceived signal %d. Do you want to terminate the program? (y/n): ", sig);
    scanf(" %c", &response);
    if (response == 'y' || response == 'Y') {
        // Απελευθέρωση των πόρων
        if (sem_close(sem) != 0) {
            perror("Failed to close semaphore");
            exit(1);
        }
        if (sem_unlink("/file_sem") != 0) {
            perror("Failed to unlink semaphore");
            exit(1);
        }
        if (pthread_mutex_destroy(&sum_mutex) != 0) {
            perror("Failed to destroy mutex");
            exit(1);
        }
        exit(sig);
    }
}

```

```

arvi@07:19:56:OS_assignment$ ./a.out
File written successfully by Process 1
Thread 0 processed 25 lines. Local sum: 62065
Thread 1 processed 25 lines. Local sum: 31799
Thread 2 processed 25 lines. Local sum: 12307
Thread 3 processed 25 lines. Local sum: 40525
Total sum of all numbers: 146696
^C
Received signal 2. Do you want to terminate the program? (y/n): y
arvi@07:20:07:OS_assignment$ echo $?
2

```

```

arvi@07:24:29:OS_assignment$ ./a.out
File written successfully by Process 1
Thread 0 processed 25 lines. Local sum: 57305
Thread 2 processed 25 lines. Local sum: 20814
Thread 3 processed 25 lines. Local sum: 48271
Thread 1 processed 25 lines. Local sum: 60630
Total sum of all numbers: 187020

Received signal 15. Do you want to terminate the program? (y/n): y
arvi@07:25:20:OS_assignment$ echo $?
15

```

Δημιουργία νημάτων και πέρασμα παραμέτρων

Τα νήματα δημιουργούνται με τη χρήση της συνάρτησης `pthread_create`. Αυτή η συνάρτηση λαμβάνει ως παραμέτρους έναν δείκτη σε μεταβλητή τύπου `pthread_t`, τις ιδιότητες του νήματος, την συνάρτηση που θα εκτελέσει το εκάστοτε νήμα, και έναν δείκτη σε δεδομένα που θα περαστούν ως παράμετροι στη συνάρτηση του νήματος. Οι παράμετροι περνιούνται στη συνάρτηση του νήματος μέσω ενός δείκτη σε δεδομένα. Αυτός ο δείκτης είναι τύπου `void*`, και μπορεί να δείχνει σε οποιοδήποτε τύπο δεδομένων.

Screenshots

```
// Δημιουργία των νημάτων
for (int i = 0; i < NUM_THREADS; i++) {
    thread_ids[i] = i;
    pthread_create(&threads[i], NULL, thread_function, (void*)&thread_ids[i]);
}
```

Χρησιμοποιήσα πίνακα για να αποφευχθούν τα race condition, εξασφαλίζοντας έτσι ότι κάθε νήμα θα έχει το δικό της αναγνωριστικό.

Συγχρονισμός νημάτων και σωστή διαδικασία μέτρησης αποτελέσματος

Ο συγχρονισμός των νημάτων και η διαδικασία μέτρησης του αποτελέσματος στο πρόγραμμά σας επιτυγχάνεται με τη χρήση ενός `mutex` και ενός σημαφόρου. Κάθε νήμα διαβάζει αριθμούς από ένα αρχείο και υπολογίζει το τοπικό του άθροισμα. Για να προσθέσει το τοπικό άθροισμα στο συνολικό άθροισμα (`total_sum`), το νήμα κλειδώνει τον `mutex` (`pthread_mutex_lock(&sum_mutex)`), ενημερώνει το συνολικό άθροισμα και στη συνέχεια ξεκλειδώνει τον `mutex` (`pthread_mutex_unlock(&sum_mutex)`), εξασφαλίζοντας ότι μόνο ένα νήμα μπορεί να ενημερώσει το συνολικό άθροισμα κάθε φορά. Η διαδικασία `process2` δημιουργεί τα νήματα και περιμένει την ολοκλήρωσή τους με τη χρήση της `pthread_join`, ενώ ο σημαφόρος (`sem_wait(sem)`) χρησιμοποιείται για να συγχρονίσει την έναρξη της `process2` με την ολοκλήρωση της `process1`. Με αυτόν τον τρόπο, εξασφαλίζεται ότι τα νήματα εκτελούνται με ασφάλεια και το συνολικό άθροισμα υπολογίζεται σωστά.

Screenshots

Ομαλή εκτέλεση προγράμματος, error handling, τεκμηρίωση

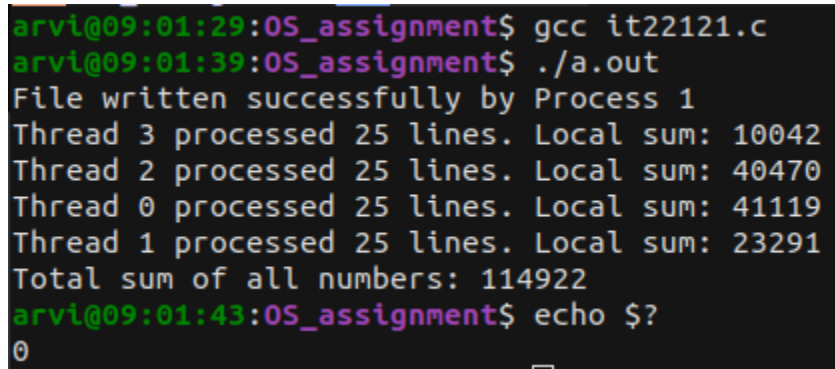
Το πρόγραμμα εκτελείται ομαλά και έχει την αναμενόμενη συμπεριφορά. Οι πόροι αποδεσμεύονται κανονικά και δεν υπάρχουν διεργασίες ζόμπι.

Οι περιπτώσεις στις οποίες έκανα έλεγχο για σφάλματα είναι οι παρακάτω:

`sem_close()`, `sem_unlink()`, `pthread_mutex_destroy()`, `fopen()`, `sem_post()`, `sem_wait()`,
`sem_open()`, `pthread_mutex_init()`, `fork()`, `waitpid()`

Ίσως και οι συναρτήσεις `pthread_create()` και `pthread_join()` να χρειάζονται έλεγχο για σφάλματα.

Screenshots



```
arvi@09:01:29:OS_assignment$ gcc it22121.c
arvi@09:01:39:OS_assignment$ ./a.out
File written successfully by Process 1
Thread 3 processed 25 lines. Local sum: 10042
Thread 2 processed 25 lines. Local sum: 40470
Thread 0 processed 25 lines. Local sum: 41119
Thread 1 processed 25 lines. Local sum: 23291
Total sum of all numbers: 114922
arvi@09:01:43:OS_assignment$ echo $?
0
```

Ερωτήσεις εκφώνησης

Προσοχή στο συγχρονισμό μέσω σημαφόρων των 2 διεργασιών. Πού πρέπει να δηλώνεται ο σημαφόρος για να είναι προσβάσιμος και γνωστός και στις 2 διεργασίες; Σε τι τιμή θα πρέπει να αρχικοποιείται και τι πράξεις να κάνει πάνω του κάθε διεργασία;

Ο σημαφόρος πρέπει να δηλώνεται πριν την δημιουργία της δεύτερης διεργασίας, ώστε να είναι προσβάσιμος και από τις δύο διεργασίες. Επίσης, πρέπει να αρχικοποιείται με 0 όπως αναφέρεται [εδώ](#). Η ενέργεια που θα κάνει η πρώτη διεργασία είναι να τον απελευθερώσει όταν ολοκληρώσει την εγγραφή και η δεύτερη εργασία αναμένει μέχρι να τελειώσει η πρώτη.

Χρειάζεται να διαμοιράζονται αυτές οι δύο διεργασίες κάποιον file descriptor;

Όχι, δεν χρειάζεται. Οι δυο διεργασίες συγχρονίζονται μέσω ενός σημαφόρου όπως αναφέρθηκε [παραπάνω](#), επίσης το πρόγραμμα γράφει στο αρχείο μόνο στην αρχή της εκτέλεσης του και δεν ξανά γράφει σε αυτό. Στην περίπτωση που στην υλοποίηση χρειαζόταν το πρόγραμμα να ξανά γράφει στο αρχείο έπειτα από την αρχική εγγραφή, τότε πιστεύω θα ήταν απαραίτητος ο διαμοιρασμός κάποιου file descriptor και η υλοποίηση με pipe.

Χρειάζεται να μοιράζονται όλα τα νήματα της 2ης διεργασίας-αναγνώστη κάποιον file descriptor ανάγνωσης; Χρειάζεται προστασία αυτός από ταυτόχρονη πρόσβαση;

Στην δική μου υλοποίηση τα νήματα δεν μοιράζονται κάποιον file descriptor και κάθε νήμα ανοίγει το αρχείο ανεξάρτητα. Ωστόσο, πιστεύω πως θα μπορούσε η process2 να ανοίγει το αρχείο μια φορά και να περνά τον file descriptor στα νήματα. Στην περίπτωση αυτή, χρειάζεται προστασία από ταυτόχρονη πρόσβαση των νημάτων στον file descriptor για να αποφευχθούν οι συνθήκες ανταγωνισμού.

Προσοχή στην ταυτόχρονη πρόσβαση στον κεντρικό πίνακα της μέτρησης. Είναι κρίσιμη περιοχή; Χρειάζεται προστασία; Δικαιολογήστε την απάντησή σας

Ναι, η ταυτόχρονη πρόσβαση στον κεντρικό πίνακα είναι κρίσιμη περιοχή και χρειάζεται προστασία. Καθώς, πολλαπλά νήματα προσπαθούν να ενημερώσουν την κοινή μεταβλητή total_sum, υπάρχει κίνδυνος race conditions, όπου οι τιμές μπορεί να ενημερωθούν με λάθος τρόπο ή να χαθούν ενημερώσεις.

Γενικά Σχόλια/Παρατηρήσεις

Για την υλοποίηση των εργασιών παρακολούθησα τα εργαστήρια που είναι διαθέσιμα στο [youtube](https://www.youtube.com/watch?v=ArviHoxha7). Η εργασία ήταν αρκετά βατή αν παρακολουθήσει και κατανοήσει κάποιος τα βίντεο.

Github repo: <https://github.com/ArviHoxha7/Operating-Systems>

Με δυσκόλεψε / δεν υλοποίησα

Όταν ολοκλήρωσα την εργασία κατάλαβα πως τα νήματα θα μπορούσαν να μοιράζονται κάποιον file descriptor, ώστε να μην χρειάζεται το κάθε νήμα να ανοίγει το αρχείο. Η υλοποίηση αυτή, πιστεύω θα ήταν πιο γρήγορη και αποδοτική.

Συνοπτικός Πίνακας

2η Εργασία		
Λειτουργία	Υλοποιήθηκε (ΝΑΙ/ΟΧΙ/ΜΕ ΡΙΚΩΣ)	Συνοπτικές Παρατηρήσεις
Βασική διεργασία η οποία ελέγχει τη δημιουργία των κατάλληλων διεργασιών και τον έλεγχο του προγράμματος	ΝΑΙ	
Συγχρονισμός των 2 διεργασιών εγγραφής ανάγνωσης	ΝΑΙ	
Διαχείριση σημάτων	ΝΑΙ	
Δημιουργία νημάτων και πέρασμα παραμέτρων	ΝΑΙ	
Συγχρονισμός νημάτων και σωστή διαδικασία μέτρησης αποτελέσματος	ΝΑΙ	Δεν το υλοποίησα με διαμοιρασμό κάποιου file descriptor
Ομαλή εκτέλεση προγράμματος, error handling, τεκμηρίωση	ΝΑΙ	