Λειτουργικά Συστήματα

Ομάδα D10

Διονύσης Ζήνδρος
Πέτρος Αγγελάτος

Άσκηση 2

Ερωτήσεις

1.1 Ερωτήσεις

1. Τα παιδιά της θα μείνουν ορφανά με αποτέλεσμα να υιοθετηθούν από την init.
2. Θα εμφανιστεί και η γονεϊακή διεργασία του Α, δηλαδή το πρόγραμμα που έφτιαξε το πρώτο παιδί.
3. Για να αποφεύγονται τα fork bombs.

1.2 Ερωτήσεις

1. Τα μηνύματα εμφανίζονται με τη σειρά που γίνονται τα forks. Δηλαδή πρώτα το μήνυμα της ρίζας, στη συνέχεια των παιδιών της ρίζας και ούτω καθεξής σε breadth-first σειρά.

1.3 Ερωτήσεις

1. Δεν χρειάζεται να περιμένουμε.
2. Τα παιδιά μπορεί να μην έχουν προλάβει να κάνουν raise SIGSTOP όταν ο πατέρας τους στείλει SIGCONT. Αν συμβεί αυτό, το SIGCONT θα αγνοηθεί και στη συνέχεια το παιδί θα μπει σε αναμονή από την οποία δε θα βγει ποτέ.

1.4 Ερωτήσεις

1. Χρησιμοποιούμε ένα σωλήνα για κάθε σχέση πατέρα-παιδιού. Δηλαδή κάθε διεργασία έχει ένα σωλήνα για να επικοινωνεί με τον πατέρα της και δύο σωλήνες για να επικοινωνεί με καθένα από τα παιδιά της, αν υπάρχουν. Αυτό είναι απαραίτητο για δύο λόγους. Πρώτον, σε περίπτωση μεγάλων αριθμών, το write κάθε διεργασίας-παιδιού ενδέχεται να μην γίνει ατομικά και έτσι τα αποτελέσματα να μπερδευτούν. Δεύτερον, σε περίπτωση που η πράξη δεν είναι αντιμεταθετική (π.χ. διαίρεση), οι δύο διεργασίες-παιδιά δεν θα ολοκληρωθούν απαραίτητα με την σειρά που κλήθηκαν, με αποτέλεσμα να μην μπορούμε να ξεχωρίσουμε τα δύο αποτελέσματα στον πατέρα.
2. Θα είναι πιο γρήγορη.

Πηγαίος κώδικας

ask2-fork.c

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

```
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */
void fork_procs(void)
{
    /*
     * initial process is A.
     */
    pid_t b, c, d;

    change_pname("A");
    printf("A: Hello!\n");

    printf("A: Forking child B...\n");
    b = fork();
    if (b == 0) {
        change_pname("B");
        printf("B: Hello!\n");
        printf("B: Forking child D...\n");
        d = fork();
        if (d == 0) {
            change_pname("D");
            printf("D: Hello!\n");
            printf("D: Sleeping...\n");
            sleep(SLEEP_PROC_SEC);
            printf("D: Goodbye...\n");
            exit(13);
        }
        printf("B: Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("B: Goodbye...\n");
        exit(19);
    }
    printf("A: Forking child C...\n");
    c = fork();
```

```
    if (c == 0) {
        change_pname("C");
        printf("C: Hello!\n");
        printf("C: Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("C: Goodbye...\n");
        exit(17);
    }

    printf("A: Sleeping...\n");
    sleep(SLEEP_PROC_SEC);
    printf("A: Goodbye...\n");
    exit(16);
}

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */
int main(void)
{
    pid_t pid;
    int status;

    /* Fork root of process tree */
    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
        /* Child */
        fork_procs();
        exit(1);
    }

    /*
     * Father
     */
```

```
    /* for ask2-signals */
    /* wait_for_ready_children(1); */

    /* for ask2-{fork, tree} */
    sleep(SLEEP_TREE_SEC);

    /* Print the process tree root at pid */
    show_pstree(pid);

    /* for ask2-signals */
    /* kill(pid, SIGCONT); */

    /* Wait for the root of the process tree to terminate */
    pid = wait(&status);
    explain_wait_status(pid, status);

    return 0;
}
```

ask2-tree.c

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

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

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

void fork_procs(struct tree_node* node)
{
    pid_t child;
    int i;

    change_pname(node->name);
    printf("%s: Hello! I have %i children.\n", node->name, node-
>nr_children);
    for (i = 0; i < node->nr_children; ++i) {
        printf("%s: Forking child %s...\n", node->name, node-
>children[i].name);
        child = fork();
        if (child == 0) {
```

```
            fork_procs(&node->children[i]);
        }
    }
    printf("%s: Sleeping...\n", node->name);
    for (i = 0; i < node->nr_children; ++i) {
        wait(NULL);
    }
    if (node->nr_children == 0) {
        sleep(SLEEP_PROC_SEC);
    }
    printf("%s: Goodbye...\n", node->name);
    exit(0);
}

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 */
int main(int argc, char** argv)
{
    pid_t pid;
    int status;
    struct tree_node *root;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
        exit(1);
    }

    root = get_tree_from_file(argv[1]);
    printf("Constructing the following process tree:\n");
    print_tree(root);

    /* Fork root of process tree */
    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
        /* Child */
        fork_procs(root);
        exit(1);
    }

    sleep(SLEEP_TREE_SEC);
```

```
    /* Print the process tree root at pid */
    show_pstree(pid);

    /* Wait for the root of the process tree to terminate */
    pid = wait(&status);
    explain_wait_status(pid, status);

    return 0;
}
```

ask2-dfs.c

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

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

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

void fork_procs(struct tree_node node)
{
    pid_t child;
    int i;
    pid_t* children;

    children = (pid_t*)malloc(node.nr_children * sizeof(pid_t));
    change_pname(node.name);
    printf("%s: Hello! I have %i children.\n", node.name, node.nr_children);
    for (i = 0; i < node.nr_children; ++i) {
        printf("%s: Forking child %s...\n", node.name, node.children[i].name);
        child = fork();
        if (child == 0) {
            fork_procs(node.children[i]);
            // this point is never reached
            printf("This should never be displayed.\n");
        }
        children[i] = child;
    }
    wait_for_ready_children(node.nr_children);
```

```
    printf("%s: Waiting for SIGSTOP...\n", node.name);
    raise(SIGSTOP);
    printf("%s: Processed.\n", node.name);
    for (i = 0; i < node.nr_children; ++i) {
        printf("%s: Killing child %s...\n", node.name, node.children[i].name);
        kill(children[i], SIGCONT);
        wait(NULL);
    }
    // printf("%s: Sleeping...\n", node.name);
    // sleep(SLEEP_PROC_SEC);
    printf("%s: Goodbye...\n", node.name);
    exit(0);
}

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 */
int main(int argc, char** argv)
{
    pid_t pid;
    int status;
    struct tree_node *root;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
        exit(1);
    }

    root = get_tree_from_file(argv[1]);
    printf("Constructing the following process tree:\n");
    print_tree(root);

    /* Fork root of process tree */
    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
        /* Child */
        fork_procs(*root);
        exit(1);
    }

    sleep(SLEEP_TREE_SEC);
```

```
    /* Print the process tree root at pid */
    show_pstree(pid);

    wait_for_ready_children(1);
    kill(pid, SIGCONT);

    /* Wait for the root of the process tree to terminate */
    pid = wait(&status);
    explain_wait_status(pid, status);

    return 0;
}
```

ask2-exp.c

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

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

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

const int INT_LEN = sizeof( int );

int do_write(int fd, int data)
{
    int bytes_written = 0;
    int status;

    while (bytes_written < INT_LEN) {
        status = write(fd, (void*)(&data) + bytes_written, INT_LEN -
bytes_written);
        if ( status < 0 ) {
            return -1;
        }
        bytes_written += status;
    }
    return 0;
}
```

```
int do_read(int fd, int *data)
{
    int bytes_read;
    int status;

    while (bytes_read < INT_LEN) {
        status = read(fd, (void*)(data) + bytes_read, INT_LEN - bytes_read);
        if (status < 0) {
            return -1;
        }
        bytes_read += status;
    }

    return 0;
}

void fork_procs(struct tree_node node, int out)
{
    pid_t child;
    pid_t children[2];
    int i;
    int f[2][2];
    int a, b;
    int result;

    change_pname(node.name);
    printf("%s: Hello! I have %i children.\n", node.name, node.nr_children);
    if (node.nr_children == 0) {
        do_write(out, atoi(node.name));
        exit(0);
    }
    for (i = 0; i < node.nr_children; ++i) {
        printf("%s: Forking child %s...\n", node.name, node.children[i].name);
        pipe(f[i]);
        child = fork();
        if (child < 0) {
            perror("fork_procs: fork");
            exit(1);
        }
        if (child == 0) {
            close(f[i][0]);
            fork_procs(node.children[i], f[i][1]);
            // this point is never reached
        }
        children[i] = child;
        close(f[i][1]);
```

```
    }

    do_read(f[0][0], &a);
    do_read(f[1][0], &b);

    switch (node.name[0]) {
        case '+':
            result = a + b;
            break;
        case '*':
            result = a * b;
            break;
    }
    do_write(out, result);

    printf("%s: Goodbye...\n", node.name);
    exit(0);
}

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 */
int main(int argc, char** argv)
{
    pid_t pid;
    int status;
    struct tree_node *root;
    int f[2];
    int result;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
        exit(1);
    }

    root = get_tree_from_file(argv[1]);
    printf("Constructing the following process tree:\n");
    print_tree(root);

    /* Fork root of process tree */
    pipe(f);
    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
```

```
    }
    if (pid == 0) {
        /* Child */
        close(f[0]);
        fork_procs(*root, f[1]);
        exit(1);
    }
    close(f[1]);
    do_read(f[0], &result);

    printf("Final result: %i\n", result);

    /* Print the process tree root at pid */
    // show_pstree(pid);

    /* Wait for the root of the process tree to terminate */
    pid = wait(&status);
    explain_wait_status(pid, status);

    return 0;
}
```

Output προγραμμάτων

```
oslabd10@lesvos:~/ntua-os/ex2$ ./ask2-fork
A: Hello!
A: Forking child B...
A: Forking child C...
B: Hello!
B: Forking child D...
A: Sleeping...
B: Sleeping...
C: Hello!
C: Sleeping...
D: Hello!
D: Sleeping...


A(7203)-+-B(7204)---D(7206)
        `-C(7205)


A: Goodbye...
B: Goodbye...
C: Goodbye...
D: Goodbye...
My PID = 7202: Child PID = 7203 terminated normally, exit status = 16
```

```
oslabd10@lesvos:~/ntua-os/ex2$ ./ask2-tree proc.tree
Constructing the following process tree:
A
        B
                E
                F
        C
        D
A: Hello! I have 3 children.
A: Forking child B...
A: Forking child C...
A: Forking child D...
B: Hello! I have 2 children.
B: Forking child E...
A: Sleeping...
D: Hello! I have 0 children.
D: Sleeping...
C: Hello! I have 0 children.
C: Sleeping...
B: Forking child F...
E: Hello! I have 0 children.
E: Sleeping...
B: Sleeping...
F: Hello! I have 0 children.
F: Sleeping...


A(7212)-+-B(7213)-+-E(7216)
        |       `-F(7217)
        |-C(7214)
        `-D(7215)


D: Goodbye...
C: Goodbye...
F: Goodbye...
E: Goodbye...
B: Goodbye...
A: Goodbye...
My PID = 7211: Child PID = 7212 terminated normally, exit status = 0
oslabd10@lesvos:~/ntua-os/ex2$ ./ask2-dfs proc.tree
Constructing the following process tree:
A
        B
                E
                F
        C
```

```
      D
A: Hello! I have 3 children.
A: Forking child B...
A: Forking child C...
B: Hello! I have 2 children.
B: Forking child E...
A: Forking child D...
B: Forking child F...
C: Hello! I have 0 children.
C: Waiting for SIGSTOP...
E: Hello! I have 0 children.
E: Waiting for SIGSTOP...
My PID = 7222: Child PID = 7224 has been stopped by a signal, signo = 19
My PID = 7223: Child PID = 7225 has been stopped by a signal, signo = 19
D: Hello! I have 0 children.
D: Waiting for SIGSTOP...
My PID = 7222: Child PID = 7226 has been stopped by a signal, signo = 19
F: Hello! I have 0 children.
F: Waiting for SIGSTOP...
My PID = 7223: Child PID = 7227 has been stopped by a signal, signo = 19
B: Waiting for SIGSTOP...
My PID = 7222: Child PID = 7223 has been stopped by a signal, signo = 19
A: Waiting for SIGSTOP...


A(7222)-+-B(7223)-+-E(7225)
        |         `-F(7227)
        |-C(7224)
        `-D(7226)


A: Processed.
A: Killing child B...
B: Processed.
B: Killing child E...
E: Processed.
E: Goodbye...
B: Killing child F...
F: Processed.
F: Goodbye...
B: Goodbye...
A: Killing child C...
C: Processed.
C: Goodbye...
A: Killing child D...
D: Processed.
D: Goodbye...
```

```
A: Goodbye...
My PID = 7221: Child PID = 7222 terminated normally, exit status = 0
oslabd10@lesvos:~/ntua-os/ex2$ ./ask2-expr expr.tree
Constructing the following process tree:
+
      10
      *
            +
                  5
                  7
            4
+: Hello! I have 2 children.
+: Forking child 10...
+: Forking child *...
10: Hello! I have 0 children.
*: Hello! I have 2 children.
*: Forking child +...
*: Forking child 4...
+: Hello! I have 2 children.
+: Forking child 5...
+: Forking child 7...
7: Hello! I have 0 children.
5: Hello! I have 0 children.
4: Hello! I have 0 children.
+: Goodbye...
*: Goodbye...
+: Goodbye...
Final result: 58


+(7234)


My PID = 7233: Child PID = 7234 terminated normally, exit status = 0
```