ΚΩΤΣΗΣ ΣΤΑΘΗΣ 03115408
ΡΑΓΚΟΥΣΗΣ ΗΛΙΑΣ 03117897

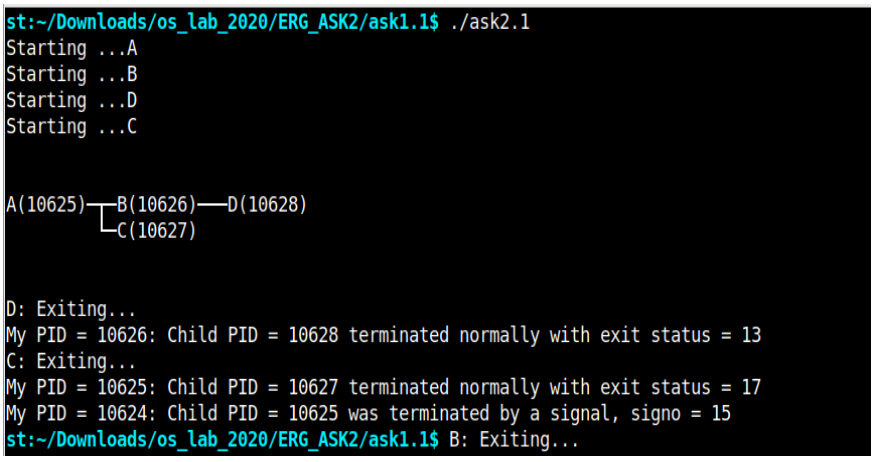
Λειτουργικά Συστήματα
6ο εξάμηνο, Ακαδημαϊκή περίοδος 2019-2020

Άσκηση 2:
Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

1.1
Δημιουργία δεδομένου δέντρου διεργασιών

ΕΡΩΤΗΣΕΙΣ 1)

Αν κάνουμε kill την parent process A παρατηρούμε ότι οι διεργασίες παιδιά της (εδώ η B γιατί οι άλλες είχαν τερματιστεί πριν το kill (signal) στην A) γίνονται παιδιά της init η οποία περιμένει τον τερματισμό τους περιοδικά για να απελευθερώσει τα resources τους



2)Αν κάνουμε show_pstree(getpid()) παρατηρούμε επιπλέον την διεργασία Ask2.1 που εκκίνησε τις διεργασίες του δέντρου μας (ρίζα) καθώς με παιδιά το tree μας το sh (shell) και την pstree.Το shell έχει ως παιδί την pstree εφόσον παράγει αυτή την διεργασία.

```
st:~/Downloads/os_lab_2020/ERG_ASK2/ask1.1$ ./ask2.1
Starting ...A
Starting ...B
Starting ...D
Starting ...C


ask2.1(17046)─┬─A(17047)─┬─B(17048)──D(17050)
              │          └─C(17049)
              └─sh(17051)──pstree(17052)


D: Exiting...
My PID = 17048: Child PID = 17050 terminated normally with exit status = 13
C: Exiting...
My PID = 17047: Child PID = 17049 terminated normally with exit status = 17
B: Exiting...
My PID = 17047: Child PID = 17048 terminated normally with exit status = 19
A: Exiting...
My PID = 17046: Child PID = 17047 terminated normally with exit status = 16
```

3)Ο διαχειριστής θέτει όρια ώστε να περιορίσει τον χρήστη.Αν ο χρήστης μπορεί να δημιουργεί πολλές διεργασίες τότε χωρίς την χρήση κατάλληλου scheduler ,πιθανόν να έχουμε διεργασίες που μονομερούν την χρήση του επεξεργαστή και άλλες που δεν εκτελούνται.Επίσης έχουμε πιθανότητα για deadlocks και ανακύπτουν κίνδυνοι ασφαλείας .Θα μπορούσε ένας κακόβουλος χρήστης να κάνει συνεχώς fork για να εξαντλήσει τους υπολογιστικούς πόρους του υπολογιστή.

1.2
Δημιουργία αυθαίρετου δέντρου διεργασιών

```
st:~/Downloads/os_lab_2020/ERG_ASK2/ask1.2$ ./ask1.2 tree_file2
A
        B
                E
                F
        C
        D
Process A is starting
Process B is starting
Process C is starting
Process D is starting
Process E is starting
Process F is starting


A(4303)─┬─B(4304)─┬─E(4307)
        │         └─F(4308)
        ├─C(4305)
        └─D(4306)


C: Exiting...
D: Exiting...
E: Exiting...
F: Exiting...
My PID = 4303: Child PID = 4305 terminated normally, exit status = 1
My PID = 4303: Child PID = 4306 terminated normally, exit status = 1
My PID = 4304: Child PID = 4307 terminated normally, exit status = 1
My PID = 4304: Child PID = 4308 terminated normally, exit status = 1
B: Exiting...
My PID = 4303: Child PID = 4304 terminated normally, exit status = 1
A: Exiting...
My PID = 4302: Child PID = 4303 terminated normally, exit status = 1
st:~/Downloads/os_lab_2020/ERG_ASK2/ask1.2$
```

ΕΡΩΤΗΣΕΙΣ 1)

Τα μηνύματα έναρξης και τερματισμού εμφανίζονται
με τυχαία σειρά εφόσον η ανδρομική συνάρτηση δεν ελέγχει τον τρόπο εμφάνισης των μηνυμάτων .Στην συνέχεια κοιμούνται εμφανίζουμε το δέντρο ,ξυπνάνε και τερματίζουν,πρώτα τα φύλλα και μετά οι γονείς προς την ρίζα.

1.3
Αποστολή και χειρισμός σημάτων

```
st:~/Downloads/os_lab_2020/ERG_ASK2/ask1.3$ ./ask2-signals tree_file2
A
        B
                E
                F
        C
        D
PID = 14617, name A, starting...
PID = 14619, name C, starting...
PID = 14620, name D, starting...
PID = 14618, name B, starting...
PID = 14621, name E, starting...
PID = 14622, name F, starting...
My PID = 14617: Child PID = 14619 has been stopped by a signal, signo = 19
My PID = 14617: Child PID = 14620 has been stopped by a signal, signo = 19
My PID = 14618: Child PID = 14621 has been stopped by a signal, signo = 19
My PID = 14618: Child PID = 14622 has been stopped by a signal, signo = 19
My PID = 14617: Child PID = 14618 has been stopped by a signal, signo = 19
My PID = 14616: Child PID = 14617 has been stopped by a signal, signo = 19


A(14617)─┬─B(14618)─┬─E(14621)
         │          └─F(14622)
         ├─C(14619)
         └─D(14620)


PID = 14617, name A, waking up ...
PID = 14618, name B, waking up ...
PID = 14621, name E, waking up ...
Process E is exiting with pid:14621
My PID = 14618: Child PID = 14621 terminated normally, exit status = 4
PID = 14622, name F, waking up ...
Process F is exiting with pid:14622
My PID = 14618: Child PID = 14622 terminated normally, exit status = 4
Process B is exiting with pid:14618
My PID = 14617: Child PID = 14618 terminated normally, exit status = 4
PID = 14619, name C, waking up ...
Process C is exiting with pid:14619
My PID = 14617: Child PID = 14619 terminated normally, exit status = 4
PID = 14620, name D, waking up ...
Process D is exiting with pid:14620
My PID = 14617: Child PID = 14620 terminated normally, exit status = 4
Process A is exiting with pid:14617
My PID = 14616: Child PID = 14617 terminated normally, exit status = 4
```

ΕΡΩΤΗΣΕΙΣ

1)Η sleep δεν μας δίνει την δυνατότητα να καθορίζουμε πλήρως πότε θα ξυπνήσει μια διεργασία και θα συνεχίσει ενώ με τα σήματα πετυχαίνουμε την ασύγχρονη εκτέλεσή τους .Η sleep οδηγεί στην αδράνεια την συγκεκριμένη διεργασία και μπορεί κάποια άλλη να εκτελεστεί στην θέση της για το χρονικό διάστημα που κοιμάται ,μπορεί όμως να προκύπτει overhead και επιπλέον το λειτουργικό να δίνει priority σε εκείνες που ξυπνάνε οπότε να χαλάσει ο συγχρονισμός επειδή μια πληρε προτεραιότητα ενώ δεν έπρεπε.Με τα signals καθορίζουμε πλήρως πότε θα ξαναξεκινήσει αλλά και υπό ποιες συνθήκες,π.χ σε ποιο σήμα θα απαντήσει και ποιο θα αγνοήσει.

2)Η wait_for_ready_children() ορίζει ότι ο γονιός θα περιμένει τα παιδιά του να σταματήσουν και να λάβει το status τους πριν συνεχίσει την δράση της .Εδώ βοηθάει ώστε να περιμένει τα παδιά να σταματήσουν μόλις τους στείλει το kill (pid) μήνυμα.Έτσι στην συνέχεια μπορεί να σταματήσει η ίδια.Αν δεν το χρησιμοποιούσαμε θα είχαμε το πρόβλημα παιδιά να μην σταματάνε πριν τον γονέα και έτσι ίσως να αγνοούσαν κάποιο sigcont στην πορεία το οποίο θα ερχόταν π.χ πριν το sigstop .Έτσι θα μπορούσαν κάποια παιδιά να σταματήσουν και να μην μπορούν να ξυπνήσουν.

1.4
Παράλληλος υπολογισμός αριθμητικής έκφρασης

```
t:~/Downloads/os_lab_2020/ERG_ASK2/ask1.4$ ./ask1.4_v3 expr.tree
-
       10
       *
              +
                     5
                     7
              4
0 :exiting
y PID = 26883: Child PID = 1 terminated normally, exit status = 1
 :exiting
 :exiting
 :exiting
y PID = 26885: Child PID = 1 terminated normally, exit status = 1
y PID = 26886: Child PID = 1 terminated normally, exit status = 1
y PID = 26886: Child PID = 1 terminated normally, exit status = 1
he answer of << 5 + 7 >> is 12
 :exiting
y PID = 26885: Child PID = 1 terminated normally, exit status = 1
he answer of << 12 * 4 >> is 48
 :exiting
y PID = 26883: Child PID = 1 terminated normally, exit status = 1
he answer of << 10 + 48 >> is 58
 :exiting
esult is :58
y PID = 26882: Child PID = 26883 terminated normally, exit status = 1
t:~/Downloads/os_lab_2020/ERG_ASK2/ask1.4$
```

Ερωτήσεις:

1)Μπορούμε να χρησιμοποιήσουμε μία σωλήνωση εφόσον τα παιδιά γράφουν στο άκρο εγγραφής , αφού κλείσουν το άκρο ανάγνωσης και ο γονέας κλείνει το άκρο εγγραφής και διαβάζει από το άκρο ανάγνωσης.Εδώ οι πράξεις είναι μιας φοράς και το αποτέλεσμα

προωθείται προς τους γονείς .Για πράξεις όπου έχει σημασία ποιος τελεστέος είναι πρώτος και ποιος 2ος χρειάζομαι περισσότερα Pipes εφόσον η μια κατεύθυνση δεν αρκεί.

2)Πέρα από το γεγονός ότι η παραλληλία πιθανότατα να οδηγεί σε μεγαλύτερη χρήση και όχι αδράνεια των επεξεργαστών πετυχαίνουμε να μπορούμε να σπάμε την πράξη (μια αριθμητική έκφραση) σε πολλές και όσες δεν εξαρτώνται από άλλες υπολογίζονται παράλληλα και το αποτέλεσμα είναι έτοιμο όταν μια άλλη διεργασία το χρειαστεί.Έτσι έχουμε αποδοτικότερο-ταχύτερο υπολογισμό.

ΚΩΔΙΚΕΣ :
ΑΣΚ1.1

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.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(char *name, int exit_code) {
  sleep(7);
  printf("%s: Exiting...\n", name);
  exit(exit_code);
}

int main(void) {
  pid_t pid2, pid3, pid4, pid5;
  int status, status2;
  int i = 0;

  pid_t pid_all[2];
  int status_all[2];

  pid2 = fork(); // create A
  if (pid2 < 0) {
    perror("A: fork");
    exit(1);
  }

  if (pid2 == 0) { // A is running
```

```
pid3 = fork(); // create B

if (pid3 < 0) {
  perror("B: fork");
  exit(1);
}

if (pid3 == 0) {
  pid4 = fork(); // create D

  if (pid4 < 0) {
    perror("D: fork");
    exit(1);
  }
  if (pid4 == 0) { // D IS running
    change_pname("D");
    printf("Starting ...D\n");

    fork_procs("D", 13);
  }

  // B is waiting
  change_pname("B");
  printf("Starting ...B\n");
  pid4 = wait(&status2);

  explain_wait_status(pid4, status2);

  fork_procs("B", 19);
}

pid5 = fork(); // create C
if (pid5 < 0) {
  perror("C: fork");
  exit(1);
}

if (pid5 == 0) {
  sleep(2);
  change_pname("C");
  printf("Starting ...C\n");

  fork_procs("C", 17);
}

// A should wait for both children B and C to finish
change_pname("A");
printf("Starting ...A\n");
while ((pid_all[i] = wait(&status_all[i])) > 0) {
  explain_wait_status(pid_all[i], status_all[i]);
  i++;
}
```

```
    fork_procs("A", 16);
  }

  /* for ask2-{fork, tree} */

  sleep(2);
  /* Print the process tree root at pid */
  show_pstree(pid2);

  /* Wait for the root of the process tree to terminate */
  pid2 = wait(&status);
  explain_wait_status(pid2, status);

  return 0;
}
```

ΑΣΚ 1.2

```
#include "tree.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define SLEEP_PROC_SEC 10
#define SLEEP_TREE_SEC 3

// create the functions
void fork_procs(char *name, int exit_code) {
  //  change_pname(name);
  sleep(6);
  printf("%s: Exiting...\n", name);
  exit(exit_code);
}

void fork_procs2(char *name) {
  change_pname(name);
  //  sleep(3);
}

void recurNode(struct tree_node *root) {

  pid_t *pid_child = (pid_t *)malloc(root->nr_children * sizeof(pid_t));
  pid_t pid;
  int status;

  change_pname(
      root->name); // change here the name -->when the process first starts
  printf("Process %s is starting \n", root->name);
```

```
  // the root has children
  for (int i = 0; i < root->nr_children; i++) {
    pid_child[i] = fork();

    if (pid_child[i] < 0) {
      perror("error in forking children");
      exit(1);
    }

    // here we have the child (i) running
    else if (pid_child[i] == 0) {
      fork_procs2(&root->children[i].name);
      recurNode(&root->children[i]);
      // sos give the address of the next child
    }
  }

  // parent waits for all children (no zombies allowed )
  while ((pid = wait(&status)) > 0)
    explain_wait_status(pid, status);

  // no children -->that means (maybe) that it is a leaf
  fork_procs(root->name, 1);
}

int main(int argc, char *argv[]) {
  struct tree_node *root;
  pid_t pid0;
  int status0;

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

  root = get_tree_from_file(argv[1]);

  pid0 = fork();

  if (pid0 < 0) {
    perror("error in forking the recursive function process");
  }
  if (pid0 == 0) {
    print_tree(root);
    recurNode(root);
  }

  sleep(2);
  // parent process waits for the function processe
  show_pstree(pid0);
```

```
  /* Wait for the root of the process tree to terminate */
  pid0 = wait(&status0);
  explain_wait_status(pid0, status0);

  return 0;
}
```

ΑΣΚ1.3

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

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

// here implement sigaction
void wakeMe(int sig, siginfo_t *info, void *ptr) {
  if (sig == SIGCONT) {
    printf("the process with pid : %ld is awake now !", (long)getpid());
    // HERE SHOULD WAKEUP //
  }
}

void catch_sigcont() {
  static struct sigaction _sigact;
  memset(&_sigact, 0, sizeof(_sigact));

  _sigact.sa_sigaction = wakeMe;
  _sigact.sa_flags = SA_SIGINFO; // sigaction handles instead of handler

  if (sigaction(SIGCONT, &_sigact, NULL) < 0) {
    perror("sigaction");
    return 1;
  }
}

void fork_procs(char *name, int exit_code, int number, pid_t *pid_child) {
  pid_t pid2, pid3;
  int status2, status3;

  printf("PID = %ld, name %s, starting...\n", (long)getpid(), name);

  sleep(2);
  wait_for_ready_children(number);
  raise(SIGSTOP);
```

```
  // here wake up and continue
  catch_sigcont();
  printf("PID = %ld, name %s, waking up ...\n", (long)getpid(), name);

  for (int i = 0; i < number; i++) {
    kill(pid_child[i], SIGCONT);

    pid3 = wait(&status3);
    explain_wait_status(pid3, status3);
  }

  printf("Process %s is exiting with pid:%ld\n", name, (long)getpid());
  exit(4);
}

void fork_procs2(char *name) { change_pname(name); }

void recurNode(struct tree_node *root) {

  pid_t *pid_child = (pid_t *)malloc(root->nr_children * sizeof(pid_t));
  pid_t pid;
  int status;

  change_pname(
      root->name); // change here the name -->when the process first starts

  // the root has children
  for (int i = 0; i < root->nr_children; i++) {
    pid_child[i] = fork();

    if (pid_child[i] < 0) {
      perror("error in forking children");
      exit(1);
    }
    // here we have the child (i) running
    else if (pid_child[i] == 0) {
      recurNode(&root->children[i]); // sos give the address of the next child
    }
  }

  // parent waits for all children (no zombies allowed )
  fork_procs(root->name, 1, root->nr_children, pid_child);
}

int main(int argc, char *argv[]) {
  pid_t pid0;
  int status0;
  struct tree_node *root;

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

```
  }

  /* Read tree into memory */
  root = get_tree_from_file(argv[1]);

  /* Fork root of process tree */
  pid0 = fork();

  if (pid0 < 0) {
    perror("main: fork");
    exit(1);
  }
  if (pid0 == 0) {

    print_tree(root);

    recurNode(root);
  }

  wait_for_ready_children(1);

  /* Print the process tree root at pid */
  show_pstree(pid0);

  /*here send sigcont to root */
  kill(pid0, SIGCONT);

  /* Wait for the root of the process tree to terminate */
  wait(&status0);
  explain_wait_status(pid0, status0);
  return 0;
}
```

ΑΣΚ1.4

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

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

int calculate(char *name, int val1, int val2) {
  if (strcmp(name, "+") == 0)
    return val1 + val2;
  else if (strcmp(name, "*") == 0)
    return val1 * val2;
}
```

```
void recurNode(struct tree_node *root, int pfd) {

  pid_t temp_pid;
  int status, pfd_c1[2], pfd_c2[2], op1, op2, temp, i;
  char *ptr;

  change_pname(root->name);

  // creating pipes//
  if (pipe(pfd_c1) < 0) {
    perror("error creating pipe\n");
    exit(1);
  }
  if (pipe(pfd_c2) < 0) {
    perror("error creating pipe\n");
    exit(1);
  }

  // passing pipes//
  for (i = 0; i < root->nr_children; i++) {
    temp_pid = fork();
    if (temp_pid < 0) {
      perror("error creating fork\n");
    } else if (temp_pid == 0) {
      if (i == 0) {
        close(pfd_c1[0]);
        recurNode(root->children + i, pfd_c1[1]);
      } else {
        close(pfd_c2[0]);
        recurNode(root->children + i, pfd_c2[1]);
      }
    }
  }

  // for leaf nodes //
  if (root->nr_children == 0) {
    // read name and convert to number//
    temp = strtol(root->name, &ptr, 10);
    if (write(pfd, &temp, sizeof(temp)) != sizeof(temp)) {

      perror("error writing leaf nodes\n");
    }
  }

  // for parents waiting to calculate//
  else {
    while (temp_pid = wait(&status) > 0)
      explain_wait_status(temp_pid, status);
    // close the writing end of child 1//
    close(pfd_c1[1]);
```

```
    if (read(pfd_c1[0], &op1, sizeof(op1)) != sizeof(op1)) {
      perror("error reading from child 1\n");
    }
    // close the writing end of child 2//
    close(pfd_c2[1]);

    if (read(pfd_c2[0], &op2, sizeof(op2)) != sizeof(op2)) {
      perror("error reading from child 2\n");
    }
    // calculate answer//
    temp = calculate(root->name, op1, op2);

    // write the answer//
    write(pfd, &temp, sizeof(temp));
    printf("The answer of << %d %s %d >> is %d\n", op1, root->name, op2, temp);
  }

  printf("%s :exiting\n", root->name);
  exit(1);
}

////////---MAIN FUNCTION---/////////
int main(int argc, char *argv[]) {
  struct tree_node *root;

  pid_t pid0;
  int status0;
  int pfd[2], result;

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

  root = get_tree_from_file(argv[1]);
  print_tree(root);

  if (pipe(pfd) < 0) {
    perror("pipe");
    exit(1);
  }

  pid0 = fork();

  if (pid0 < 0) {
    perror("error in forking the recursive function's process\n");
  }
  if (pid0 == 0) {
    close(pfd[0]);
    recurNode(root, pfd[1]);
    exit(1);
  }
```

```
  // parent reads the result//
  close(pfd[1]);
  read(pfd[0], &result, sizeof(result));
  printf("result is :%d\n", result);

  // parent process waits for the function processe

  /* Wait for the root of the process tree to terminate */
  pid0 = wait(&status0);
  explain_wait_status(pid0, status0);

  return 0;
}
```