**Λειτουργικά Συστήματα**

**Άσκηση 2**

Περράκης Γιώργος 03113511

**Άσκηση 1.1**

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\_1 1

#define SLEEP\_TREE\_SEC 3

int main(void)

{

pid\_t p\_A;

int status;

p\_A = fork();

if( p\_A < 0 ) //main creates A

{

perror("main: fork");

exit(1);

}

else if( p\_A == 0 )

{

printf("A: starting\n");

printf("A: waiting\n");

change\_pname("A");

pid\_t p\_B;

p\_B = fork();

if ( p\_B < 0 )

{

perror("A: fork");

exit(1);

}

else if ( p\_B == 0 )

{

/\*B child\*/

change\_pname("B");

printf("B: starting\n");

printf("B: waiting\n");

pid\_t p\_D;

p\_D = fork();

if( p\_D < 0 )

{

perror("B: fork");

exit(1);

}

else if(p\_D == 0)

{

/\*child D\*/

change\_pname("D");

printf("D: starting\n");

printf("D: sleeping\n");

sleep(SLEEP\_PROC\_SEC);

printf("D: exiting\n");

exit(13);

}

sleep(SLEEP\_1);

printf("B: waiting for D to exit\n");

p\_B = wait(&status); //B waits D

explain\_wait\_status(p\_B, status);

exit(19);

}

else

{

sleep(SLEEP\_1);

printf("A: waiting for B to exit\n");

pid\_t p\_C;

p\_C = fork();

if(p\_C < 0)

{

perror("A: fork");

exit(1);

}

else if( p\_C == 0)

{

/\*C child\*/

change\_pname("C");

printf("C: starting\n");

printf("C: sleeping\n");

sleep(SLEEP\_PROC\_SEC);

printf("C: exiting\n");

exit(17);

}

sleep(SLEEP\_1);

printf("A: waiting for C to exit\n");

p\_C = wait(&status); //A waits C

explain\_wait\_status(p\_C, status);

}

p\_B = wait(&status); //A waits B

explain\_wait\_status(p\_B, status);

printf("A: exiting\n");

exit(16);

}

sleep(SLEEP\_TREE\_SEC);

show\_pstree(p\_A); /\* Print the process tree root at pid \*/

p\_A = wait(&status);

explain\_wait\_status(p\_A, status);

return 0;

}

**Έξοδος προγράμματος:**

A: starting

A: waiting

B: starting

B: waiting

D: starting

D: sleeping

A: waiting for B to exit

C: starting

C: sleeping

B: waiting for D to exit

A: waiting for C to exit

A(16958)─┬─B(16959)───D(16960)

└─C(16961)

D: exiting

My PID = 16959: Child PID = 16960 terminated normally, exit status = 13

My PID = 16958: Child PID = 16959 terminated normally, exit status = 19

C: exiting

My PID = 16958: Child PID = 16961 terminated normally, exit status = 17

A: exiting

My PID = 16957: Child PID = 16958 terminated normally, exit status = 16

**Ερώτηση 1**

Με τον τερματισμό της διεργασίας Α, όλα τα παιδιά της θα κληρονομηθούν από την διεργασία init, η οποία βρίσκεται συνεχώς σε κατάσταση wait, περιμένοντας τα παιδιά να τερματιστούν.

**Ερώτηση 2**

Με την εκτέλεση της εντολής show\_pstree(getpid()) αντί της εντολής show\_pstree(pid), εμφανίζεται σαν ρίζα διεργασία η ask2-fork. Αυτό οφείλεται στο γεγονός ότι η μεταβλητή pid περιέχει το PID της διεργασίας Α που είναι παιδί της ask2-fork και η ρίζα του επιθυμητού δέντρου, ενώ με το getpid() παίρνουμε το PID της ask2-fork. Στο δέντρο φαίνονται οι εξής επιπλέον διεργασίες: ask2-fork ως ρίζα, η sh ως παιδί της ask2-fork και η pstree ως παιδί της sh. Ο λόγος είναι ότι η ρουτίνα show\_pstree, που καλείται από την ask2-fork, χρησιμοποιεί την διεργασία pstree.

**Ερώτηση 3**

Ο λόγος για τον οποίο ο διαχειριστής θέτει όρια για να εμποδίσει τους πολλαπλούς χρήστες από το να δημιουργούν συνέχεια διεργασίες, είναι ότι έτσι το λειτουργικό σύστημα θα προσπαθούσε να εξυπηρετήσει όλες τις διεργασίες το ίδιο και με αυτό τον τρόπο θα προκαλούνταν καθυστερήσεις και το σύστημα δεν θα ήταν αποδοτικό. Παράλληλα, με λιγότερες διεργασίες ο διαχειριστής θα είχε την δυνατότητα να διαμοιράσει πιο δίκαια τους πόρους του λειτουργικού συστήματος μεταξύ των χρηστών.

**Άσκηση 1.2**

tree-example.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <assert.h>

#include <string.h>

#include <sys/prctl.h>

#include <sys/wait.h>

#include "tree.h"

#include "proc-common.h"

#define BUFF\_SIZE 1024

#define SLEEP\_CHILD 10

#define SLEEP\_MAIN 3

int main(int argc, char \*argv[])

{

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]);

print\_tree(root);

return 0;

}

static void \_\_print\_tree(struct tree\_node \*root, int level)

{

int i;

for (i=0; i<level; i++)

printf("\t");

printf("%s\n", root->name);

for (i=0; i < root->nr\_children; i++)

{

\_\_print\_tree(root->children + i, level + 1);

}

}

void print\_tree(struct tree\_node \*root)

{

int status;

\_\_print\_tree(root, 0);

pid\_t p;

p=fork();

printf("%ld",(long)p);

if(p < 0)

{

perror("fork1\n");

exit(1);

}

if( p == 0 )

{

/\*child\*/

printf("Main: Created A\n");

proc\_tree(root, 0);

exit(14);

}

else

{

/\*father\*/

printf("Main: waiting for A to terminate\n");

sleep(SLEEP\_MAIN);

show\_pstree(p);

p=wait(&status);

}

}

static void proc\_tree(struct tree\_node \*root, int level)

{

int i,status;

change\_pname(root->name);

if(root->nr\_children == 0)

{

printf("Child: My pid id %ld and i'm sleeping\n", (long)getpid());

sleep(SLEEP\_CHILD);

return;

}

else

{

pid\_t p[root->nr\_children];

for ( i= 0; i < root->nr\_children; i++)

{

p[i] = fork();

if(p[i] < 0)

{

perror("fork\n");

exit(1);

}

if( p[i] == 0)

{

/\*child\*/

proc\_tree(root->children + i, level + 1);

printf("Child: My pid is %ld and i'm exiting\n",(long)getpid());

exit(14);

}

else

{

/\*father\*/

printf("Father: Created child with pid %ld and waiting for it to terminate\n", (long)p[i]);

}

}

for(i=0; i < root->nr\_children; i++)

{

p[i]=wait(&status);

}

printf("Father: My pid is %ld and i'm exiting\n",(long)getpid());

exit(14);

}

}

**Αρχείο Εισόδου**

A

2

B

C

B

3

D

E

P

D

0

E

0

P

1

G

G

0

C

1

F

F

0

**Έξοδος προγράμματος:**

A

B

D

E

P

G

C

F

16977Main: waiting for A to terminate

0Main: Created A

Father: Created child with pid 16978 and waiting for it to terminate

Father: Created child with pid 16979 and waiting for it to terminate

Father: Created child with pid 16980 and waiting for it to terminate

Father: Created child with pid 16981 and waiting for it to terminate

Father: Created child with pid 16982 and waiting for it to terminate

Father: Created child with pid 16983 and waiting for it to terminate

Child: My pid id 16980 and i'm sleeping

Child: My pid id 16982 and i'm sleeping

Child: My pid id 16981 and i'm sleeping

Father: Created child with pid 16984 and waiting for it to terminate

Child: My pid id 16984 and i'm sleeping

A(16977)─┬─B(16978)─┬─D(16981)

│ ├─E(16982)

│ └─P(16983)───G(16984)

└─C(16979)───F(16980)

Child: My pid is 16980 and i'm exiting

Father: My pid is 16979 and i'm exiting

Child: My pid is 16982 and i'm exiting

Child: My pid is 16981 and i'm exiting

Child: My pid is 16984 and i'm exiting

Father: My pid is 16983 and i'm exiting

Father: My pid is 16978 and i'm exiting

Father: My pid is 16977 and i'm exiting

**Ερώτηση 1**

Όπως φαίνεται και από το output του κώδικα, η σειρά των μηνυμάτων έναρξης και τερματισμού είναι τυχαία, αφού όλες οι διεργασίες τρέχουν παράλληλα. Το λειτουργικό σύστημα δίνει κάποιο χρονικό διάστημα εκτέλεσης σε κάθε διεργασία με τη σειρά, και κάποιες μπορεί να ολοκληρωθούν νωρίτερα από άλλες, έτσι δεν μπορούμε να γνωρίζουμε με βεβαιότητα τη σειρά τους.

**Άσκηση 1.3**

ask2-signals.c

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/wait.h>

#include "tree.h"

#include "proc-common.h"

void fork\_procs(struct tree\_node \*root)

{

int status;

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

change\_pname(root->name);

if(root->nr\_children == 0)

{

printf("PID = %ld, name = %s stops\n",(long)getpid(), root->name);

raise(SIGSTOP);

printf("%s: My pid is %ld and i'm exiting\n", root->name, (long)getpid());

exit(14);

}

else

{

int i;

pid\_t p[root->nr\_children];

for(i=0; i <root->nr\_children; i++)

{

p[i]=fork();

if(p[i] < 0)

{

perror("fork\n");

exit(1);

}

if(p[i] == 0)

{

/\*child\*/

fork\_procs(root->children + i);

}

}

wait\_for\_ready\_children(root->nr\_children);

printf("PID = %ld, name =%s stops\n",(long)getpid(), root->name);

raise(SIGSTOP);

for(i=0; i<root->nr\_children; i++)

{

kill(p[i],SIGCONT);

p[i]=wait(&status);

explain\_wait\_status(p[i], status);

}

printf("%s:My pid is %ld and i'm exiting\n", root->name, (long)getpid());

exit(14);

}

return;

}

int main(int argc, char \*argv[])

{

pid\_t pid;

int status;

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 \*/

pid = fork();

if (pid < 0)

{

perror("main: fork");

exit(1);

}

if (pid == 0)

{

/\* Child \*/

fork\_procs(root);

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 \*/

wait(&status);

explain\_wait\_status(pid, status);

return 0;

}

**Αρχείο Εισόδου**

A

2

B

C

B

3

D

E

P

D

0

E

0

P

1

G

G

0

C

1

F

F

0

**Έξοδος προγράμματος:**

PID = 16991, name A, starting...

PID = 16993, name C, starting...

PID = 16992, name B, starting...

PID = 16994, name F, starting...

PID = 16994, name = F stops

My PID = 16993: Child PID = 16994 has been stopped by a signal, signo = 19

PID = 16993, name =C stops

My PID = 16991: Child PID = 16993 has been stopped by a signal, signo = 19

PID = 16997, name P, starting...

PID = 16996, name E, starting...

PID = 16996, name = E stops

My PID = 16992: Child PID = 16996 has been stopped by a signal, signo = 19

PID = 16995, name D, starting...

PID = 16995, name = D stops

My PID = 16992: Child PID = 16995 has been stopped by a signal, signo = 19

PID = 16998, name G, starting...

PID = 16998, name = G stops

My PID = 16997: Child PID = 16998 has been stopped by a signal, signo = 19

PID = 16997, name =P stops

My PID = 16992: Child PID = 16997 has been stopped by a signal, signo = 19

PID = 16992, name =B stops

My PID = 16991: Child PID = 16992 has been stopped by a signal, signo = 19

PID = 16991, name =A stops

My PID = 16990: Child PID = 16991 has been stopped by a signal, signo = 19

A(16991)─┬─B(16992)─┬─D(16995)

│ ├─E(16996)

│ └─P(16997)───G(16998)

└─C(16993)───F(16994)

D: My pid is 16995 and i'm exiting

My PID = 16992: Child PID = 16995 terminated normally, exit status = 14

E: My pid is 16996 and i'm exiting

My PID = 16992: Child PID = 16996 terminated normally, exit status = 14

G: My pid is 16998 and i'm exiting

My PID = 16997: Child PID = 16998 terminated normally, exit status = 14

P:My pid is 16997 and i'm exiting

My PID = 16992: Child PID = 16997 terminated normally, exit status = 14

B:My pid is 16992 and i'm exiting

My PID = 16991: Child PID = 16992 terminated normally, exit status = 14

F: My pid is 16994 and i'm exiting

My PID = 16993: Child PID = 16994 terminated normally, exit status = 14

C:My pid is 16993 and i'm exiting

My PID = 16991: Child PID = 16993 terminated normally, exit status = 14

A:My pid is 16991 and i'm exiting

My PID = 16990: Child PID = 16991 terminated normally, exit status = 14

**Ερώτηση 1**

Η χρήση σημάτων μας επιτρέπει να καθορίσουμε πλήρως τη σειρά με την οποία αναστέλλουν και συνεχίζουν τη λειτουργία τους. Προηγουμένως, με τη sleep(), η σειρά τους ήταν τυχαία, ενώ τώρα, καθορίζουμε η σειρά με την οποία θα συνεχίσουν (SIGCONT) τη λειτουργία τους να γίνεται κατά βάθος, και έτσι να τυπωθεί κατάλληλα το δέντρο.

**Ερώτηση 2**

Αυτή η συνάρτηση έχει σκοπό να δίνει τον απαιτούμενο χρόνο να δημιουργηθούν τα παιδιά της κάθε γονικής διεργασίας και να ανασταλεί η λειτουργία τους. Εξασφαλίζει ότι θα φτιαχτούν όλα τα παιδιά, δηλαδή όλοι που στηρίζονται σε αυτή τη γονική διεργασία, και μετά αυτή με τη σειρά της θα αναστείλει τη λειτουργία της ώστε να συνεχίσει ο δικός της γονέας. Αν παραλειπόταν αυτή η συνάρτηση, τότε πριν να δημιουργηθούν τα παιδιά της ρίζας διεργασίας, η διεργασία ask2-signals θα προχωρούσε παρακάτω χωρίς να περιμένει να δημιουργηθεί ολόκληρο το δέντρο και άρα δεν θα τυπωνόταν ορθά το δέντρο των διεργασιών.

**Άσκηση 1.4**

ask2-pipes.c

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <string.h>

#include "proc-common.h"

#include "tree.h"

void fork\_procs(struct tree\_node \*root, int \*pfd)

{

int status,value1;

char \*op="+";

close(pfd[0]); // child is never going to read from father

if(root->nr\_children == 0)

{

value1 = atoi(root->name);

printf("%d\n",value1);

write(pfd[1],&value1,sizeof(int));

}

else

{

pid\_t p1,p2;

int value,value1,value2,num1,num2;

int fd[2];

if(pipe(fd) < 0)

{

perror("pipe");

exit(1);

}

p1=fork();

if(p1 < 0)

{

perror("fork");

exit(1);

}

else if(p1 == 0)

{

/\*child\*/

fork\_procs(root->children,fd);

exit(14);

}

else

{

/\*father\*/

p2 = fork();

if(p2 < 0)

{

perror("fork");

exit(1);

}

else if(p2 == 0)

{

/\*child\*/

fork\_procs(root->children +1, fd);

close(fd[1]);

exit(14);

}

close(fd[1]);

read(fd[0],&num1,sizeof(int));

read(fd[0],&num2,sizeof(int));

close(fd[0]);

p1 = wait(&status);

explain\_wait\_status(p1, status);

p2 = wait(&status);

explain\_wait\_status(p2, status);

if(!strcmp((root->name),op))

{

/\*addition\*/

value = num1 + num2;

}

else

{

/\*multiplication\*/

value = num1 \* num2;

}

printf("%d\n",value);

write(pfd[1],&value,sizeof(int));

exit(14);

}

}

}

int main(int argc, char \*argv[])

{

pid\_t p;

int status;

int value;

int fd[2];

struct tree\_node \*root;

if(argc < 2)

{

fprintf(stderr,"Usage: %s<tree\_file.\n", argv[0]);

exit(1);

}

root = get\_tree\_from\_file(argv[1]);

if(pipe(fd) < 0)

{

perror("pipe");

exit(1);

}

p = fork();

if( p < 0)

{

perror("fork");

exit(1);

}

else if(p == 0)

{

fork\_procs(root,fd);

exit(1);

}

close(fd[1]); // father is never going to write

read(fd[0],&value,sizeof(int));

printf("%d\n",value);

p = wait(&status);

return 0;

}

**Αρχείο Εισόδου**

\*

2

+

10

+

2

5

6

5

0

6

0

10

0

**Έξοδος προγράμματος:**

10

6

5

My PID = 17005: Child PID = 17008 terminated normally, exit status = 14

My PID = 17005: Child PID = 17007 terminated normally, exit status = 14

11

My PID = 17004: Child PID = 17005 terminated normally, exit status = 14

My PID = 17004: Child PID = 17006 terminated normally, exit status = 14

110

110

**Ερώτηση 1**

Κάθε διεργασία (συμπεριλαμβανομένου και της ask1\_4) απαιτεί τη δημιουργία ενός pipe, εκτός αν πρόκειται για διεργασία φύλλο. Κάθε γονιός οφείλει να δημιουργεί ένα pipe (πριν την κλήση συστήματος fork()) ώστε να την κληρονομούν τα παιδιά της με σκοπό να μπορούν να επιστρέφουν τις αποτιμήσεις τους στο γονέα τους. Αυτό είναι και το ενδεικτικό, αφού κάθε αριθμητικός τελεστής οφείλει να διαβάζει δύο φορές από το pipe (τιμές των παιδιών του) και να γράφει στο pipe μία φορά (να δίνει την αποτίμηση του στον γονέα του).

**Ερώτηση 2**

Σε ένα σύστημα πολλαπλών επεξεργαστών μπορούν να επιστρέφονται ταυτόχρονα οι αποτιμήσεις των παιδιών κάθε αριθμητικού τελεστή, αφού δεν αλληλοεξαρτώνται, και έτσι να κερδίζεις σημαντικό χρόνο. Στην περίπτωση δέντρου διεργασιών, υπολογίζονται παράλληλα οι διεργασίες, λόγω των πολλαπλών επεξεργαστών, ενώ στην περίπτωση μιας διεργασίας, πρέπει να γίνει η αποτίμηση της ενιαία από ένα επεξεργαστή, και έτσι δεν μπορούν να αξιοποιηθούν οι υπόλοιποι επεξεργαστές.