**Λειτουργικά Συστήματα**

**Άσκηση 4**

Περράκης Γιώργος 03113511

Γκανάς Αλέξανδρος 03113088

Ομάδα εργαστηρίου c06

**Άσκηση 1.1**

#include <errno.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <assert.h>

#include <stdbool.h>

#include <sys/wait.h>

#include <sys/types.h>

#include "proc-common.h"

#include "request.h"

/\* Compile-time parameters. \*/

#define SCHED\_TQ\_SEC 2 /\* time quantum \*/

#define TASK\_NAME\_SZ 60 /\* maximum size for a task's name \*/

struct procs{

int proc\_id;

pid\_t proc\_pid;

struct procs \*next;

};

struct procs \*previous;

struct procs \*pointer;

struct procs \*start\_list;

int nproc;

/\*

\* SIGALRM handler

\*/

static void sigalrm\_handler(int signum)

{

if(signum != SIGALRM)

{

printf("Caught wrong signal\n");

exit(1);

}

else

{

kill(pointer->proc\_pid,SIGSTOP);

}

}

/\*

\* SIGCHLD handler

\*/

static void sigchld\_handler(int signum)

{

pid\_t p;

int status;

for(;;)

{

if(nproc == 0)

exit(1);

else

{

p = waitpid(-1, &status, WUNTRACED | WNOHANG);

if(p < 0)

{

perror("waitpid");

exit(1);

}

if(p == 0)

{

break;

}

explain\_wait\_status(p, status);

if(WIFEXITED(status) || WIFSIGNALED(status))

{

/\*A child has died\*/

if(pointer->next != NULL) //if we are not in the last node

{

if(pointer != start\_list) //if we are not in the first node

{

previous->next = pointer->next; //connect previous with next node

free(pointer);

pointer = previous->next; //pointer points to the next node

}

else //if we are in the first node

{

start\_list = pointer->next; //start\_list points to the second node

free(pointer);

pointer = start\_list; //pointer also points to the next node

}

}

else if( pointer->next == NULL) //if we are in the last node

{

previous->next = NULL; //previous points to the last node

free(pointer);

pointer = start\_list; //pointer points to the first node

}

kill(pointer->proc\_pid, SIGCONT);

nproc--;

alarm(SCHED\_TQ\_SEC);

}

else if(WIFSTOPPED(status))

{

/\*A child has stopped due to SIGSTOP/SIGTSTP, etc \*/

if(pointer->next != NULL) //if we are not in the last node

{

pointer = pointer->next; //pointer points to the next node

if(previous->next != NULL)

{

previous = previous->next;

}

else

{

previous = start\_list;

}

}

else

{

if(previous->next != NULL)

{

previous = previous->next;

}

else

{

previous = start\_list;

}

pointer = start\_list;

}

kill(pointer->proc\_pid, SIGCONT);

alarm(SCHED\_TQ\_SEC);

}

} }

}

/\* Install two signal handlers.

\* One for SIGCHLD, one for SIGALRM.

\* Make sure both signals are masked when one of them is running.

\*/

static void install\_signal\_handlers(void)

{

sigset\_t sigset;

struct sigaction sa;

sa.sa\_handler = sigchld\_handler;

sa.sa\_flags = SA\_RESTART;

sigemptyset(&sigset);

sigaddset(&sigset, SIGCHLD);

sigaddset(&sigset, SIGALRM);

sa.sa\_mask = sigset;

if (sigaction(SIGCHLD, &sa, NULL) < 0)

{

perror("sigaction: sigchld");

exit(1);

}

sa.sa\_handler = sigalrm\_handler;

if (sigaction(SIGALRM, &sa, NULL) < 0)

{

perror("sigaction: sigalrm");

exit(1);

}

/\*

\* Ignore SIGPIPE, so that write()s to pipes

\* with no reader do not result in us being killed,

\* and write() returns EPIPE instead.

\*/

if(signal(SIGPIPE, SIG\_IGN) < 0)

{

perror("signal: sigpipe");

exit(1);

}

}

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

{

/\*

\* For each of argv[1] to argv[argc - 1],

\* create a new child process, add it to the process list.

\*/

int i;

nproc = argc - 1; /\* number of proccesses goes here \*/

pid\_t p[nproc];

struct procs \*root;

char \*newargv[] = { NULL, NULL, NULL, NULL };

char \*newenviron[] = { NULL };

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

{

/\* Creation of list\*/

if(pointer != NULL)

{

root = ( struct procs \* )malloc(sizeof(struct procs) );

pointer->next = root;

pointer = root;

pointer->next = NULL;

}

else

{

root = ( struct procs \* ) malloc(sizeof(struct procs) );

pointer = root;

previous = root;

start\_list = root;

}

p[i] = fork();

if( p[i] < 0 )

{

perror("fork");

exit(1);

}

else if ( p[i] == 0 )

{

/\* child \*/

raise(SIGSTOP);

newargv[0] = argv[i+1];

execve(argv[i+1], newargv, newenviron);

}

}

/\* Wait for all children to raise SIGSTOP before exec()ing. \*/

wait\_for\_ready\_children(nproc);

pointer = start\_list;

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

{

kill(p[i],SIGCONT);

pointer->proc\_pid = p[i];

pointer->proc\_id = i + 1;

pointer = pointer->next;

}

wait\_for\_ready\_children(nproc);

/\* Install SIGALRM and SIGCHLD handlers. \*/

install\_signal\_handlers();

pointer = start\_list;

kill(pointer->proc\_pid,SIGCONT);

alarm(SCHED\_TQ\_SEC);

if (nproc == 0)

{

fprintf(stderr, "Scheduler: No tasks. Exiting...\n");

exit(1);

}

/\* loop forever until we exit from inside a signal handler. \*/

while (pause())

;

/\* Unreachable \*/

fprintf(stderr, "Internal error: Reached unreachable point\n");

return 1;

}

**Ερωτήσεις**

1. Η συνάρτηση install\_signal\_handlers() έχει φροντίσει έτσι ώστε κάθε φορά που εκτελείται ένας handler να μπλοκάρει τα σήματα SIGALARM και SIGCHLD. Στον χρονοδρομολογητή μας αυτό συμβαίνει με το μπλοκάρισμα αυτών των σημάτων, μέσω μασκών που χρησιμοποιούνται. Παράλληλα, ένας πραγματικός χρονοδρομολογητής χώρου πυρήνα αντιμετωπίζει ανάλογα ενδεχόμενα με καταχωρητές μασκών για κάθε interrupt που προκαλείται, έτσι ώστε να μπλοκάρονται τα άλλα σήματα.
2. Κάθε φορά που ο χρονοδρομολογητής λαμβάνει σήμα SIGCHLD αναφέρεται στην διεργασία που μόλις σταμάτησε (ή τερματίστηκε). Αν λόγω εξωτερικού παράγοντα (π.χ. αποστολή SIGKILL) τερματιστεί αναπάντεχα μια οποιαδήποτε διεργασία-παιδί, τότε ναι μεν οι handlers θα το αγνοήσουν, αλλά θα προκληθεί πρόβλημα, γιατί δεν θα έχουμε αφαιρέσει τον κόμβο που αναφέρεται σε αυτή την διεργασία.
3. Ο χρονοδρομολογητής δεν θα μπορούσε να χρησιμοποιεί μόνο το σήμα SIGALARM, γιατί σε αυτή την περίπτωση όταν θα έστελνε το σήμα SIGSTOP δεν θα μπορούσε να ξέρει σίγουρα αν έγινε με επιτυχία και αν δηλαδή η διεργασία όντως σταμάτησε. Με το χειρισμό του SIGCHLD είμαστε σίγουροι ότι η διεργασία έχει σταματήσει και άρα αποφεύγεται το ενδεχόμενο να τρέχουν δύο διεργασίες ταυτόχρονα.

**Άσκηση 1.2**

#includen<errno.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <assert.h>

#include <stdbool.h>

#include <sys/wait.h>

#include <sys/types.h>

#include "proc-common.h"

#include "request.h"

/\* Compile-time parameters. \*/

#define SCHED\_TQ\_SEC 2 /\* time quantum \*/

#define TASK\_NAME\_SZ 60 /\* maximum size for a task's name \*/

#define SHELL\_EXECUTABLE\_NAME "shell" /\* executable for shell \*/

struct procs{

int proc\_id;

pid\_t proc\_pid;

char \*name;

struct procs \*next;

};

struct procs \*root;

struct procs \*previous;

struct procs \*pointer;

struct procs \*start\_list;

struct procs \*last\_p;

struct procs \*print\_p;

struct procs \*temp\_p;

int nproc;

/\* Print a list of all tasks currently being scheduled. \*/

static void sched\_print\_tasks(void)

{

printf("Process that is currently running: %d\n", pointer->proc\_id);

print\_p = start\_list;

while(print\_p->next != NULL)

{

printf("Process id: %d\n", print\_p->proc\_id);

printf("Process pid: %d\n", print\_p->proc\_pid);

if(print\_p->proc\_id != 0)

printf("Process name: %s\n", print\_p->name);

else

printf("Process name: %s\n", print\_p->name);

print\_p = print\_p->next;

}

printf("Process id: %d\n", print\_p->proc\_id);

printf("Process pid: %d\n", print\_p->proc\_pid);

if(print\_p->proc\_id != 0)

printf("Process name: %s\n", print\_p->name);

else

printf("Process name: %s\n", print\_p->name);

print\_p = start\_list;

}

/\* Send SIGKILL to a task determined by the value of its

\* scheduler-specific id.

\*/

static int sched\_kill\_task\_by\_id(int id)

{

print\_p = start\_list;

bool flag3 = true;

while(print\_p->proc\_id != id)

{

print\_p = print\_p->next;

if(flag3 == true)

{

temp\_p = start\_list;

flag3 = false;

}

else

temp\_p = temp\_p->next;

}

kill(print\_p->proc\_pid, SIGTERM);

nproc--;

temp\_p->next = print\_p->next;

if(print\_p == previous)

{

previous = temp\_p;

}

if(print\_p == pointer)

{

pointer = print\_p->next;

}

free(print\_p);

print\_p = start\_list;

return 1;

}

/\* Create a new task \*/

static void sched\_create\_task(char \*executable)

{

pid\_t p;

char \*newargv[] = { NULL, NULL, NULL, NULL };

char \*newenviron[] = { NULL };

int temp;

newargv[0] = executable;

p = fork();

if( p < 0 )

{

perror("fork");

exit(1);

}

else if ( p == 0 )

execve(executable, newargv, newenviron);

root = ( struct procs \* )malloc(sizeof(struct procs) );

last\_p->next = root;

previous = root;

temp = last\_p->proc\_id;

last\_p = root;

last\_p->next = NULL;

last\_p->proc\_id = temp + 1;

last\_p->name = executable;

last\_p->proc\_pid = p;

nproc++;

}

/\* Process requests by the shell \*/

static int process\_request(struct request\_struct \*rq)

{

switch (rq->request\_no)

{

case REQ\_PRINT\_TASKS:

sched\_print\_tasks();

return 0;

case REQ\_KILL\_TASK:

return sched\_kill\_task\_by\_id(rq->task\_arg);

case REQ\_EXEC\_TASK:

sched\_create\_task(rq->exec\_task\_arg);

return 0;

default:

return -ENOSYS;

}

}

/\*

\* SIGALRM handler

\*/

static void sigalrm\_handler(int signum)

{

if(signum != SIGALRM)

{

printf("Caught wrong signal\n");

exit(1);

}

else

{

kill(pointer->proc\_pid,SIGSTOP);

}

}

/\*

\* SIGCHLD handler

\*/

static void sigchld\_handler(int signum)

{

pid\_t p;

int status;

for(;;)

{

if(nproc == 0)

exit(1);

else

{

p = waitpid(-1, &status, WUNTRACED | WNOHANG);

if(p < 0)

{

perror("waitpid");

exit(1);

}

if(p == 0)

{

break;

}

explain\_wait\_status(p, status);

if(WIFEXITED(status) || WIFSIGNALED(status))

{

/\*A child has died\*/

if(pointer->next != NULL) //if we are not in the last node

{

if(pointer != start\_list) //if we are not in the first node

{

previous->next = pointer->next; //connect previous with next node

free(pointer);

pointer = previous->next; //pointer points to the next node

}

else //if we are in the first node

{

start\_list = pointer->next; //start\_list points to the second node

free(pointer);

pointer = start\_list; //pointer also points to the next node

}

}

else if( pointer->next == NULL) //if we are in the last node

{

last\_p = previous;

previous->next = NULL; //previous points to the last node

free(pointer);

pointer = start\_list; //pointer points to the first node

}

kill(pointer->proc\_pid, SIGCONT);

alarm(SCHED\_TQ\_SEC);

nproc--;

}

else if(WIFSTOPPED(status))

{

/\*A child has stopped due to SIGSTOP/SIGTSTP, etc \*/

if(pointer->next != NULL) //if we are not in the last node

{

pointer = pointer->next; //pointer points to the next node

if(previous->next != NULL)

{

previous = previous->next;

}

else

{

previous = start\_list;

}

}

else

{

if(previous->next != NULL)

{

previous = previous->next;

}

else

{

previous = start\_list;

}

pointer = start\_list;

}

kill(pointer->proc\_pid, SIGCONT);

alarm(SCHED\_TQ\_SEC);

}

} }

}

/\* Disable delivery of SIGALRM and SIGCHLD. \*/

static void signals\_disable(void)

{

sigset\_t sigset;

sigemptyset(&sigset);

sigaddset(&sigset, SIGALRM);

sigaddset(&sigset, SIGCHLD);

if (sigprocmask(SIG\_BLOCK, &sigset, NULL) < 0) {

perror("signals\_disable: sigprocmask");

exit(1);

}

}

/\* Enable delivery of SIGALRM and SIGCHLD. \*/

static void signals\_enable(void)

{

sigset\_t sigset;

sigemptyset(&sigset);

sigaddset(&sigset, SIGALRM);

sigaddset(&sigset, SIGCHLD);

if (sigprocmask(SIG\_UNBLOCK, &sigset, NULL) < 0) {

perror("signals\_enable: sigprocmask");

exit(1);

}

}

/\* Install two signal handlers.

\* One for SIGCHLD, one for SIGALRM.

\* Make sure both signals are masked when one of them is running.

\*/

static void install\_signal\_handlers(void)

{

sigset\_t sigset;

struct sigaction sa;

sa.sa\_handler = sigchld\_handler;

sa.sa\_flags = SA\_RESTART;

sigemptyset(&sigset);

sigaddset(&sigset, SIGCHLD);

sigaddset(&sigset, SIGALRM);

sa.sa\_mask = sigset;

if (sigaction(SIGCHLD, &sa, NULL) < 0) {

perror("sigaction: sigchld");

exit(1);

}

sa.sa\_handler = sigalrm\_handler;

if (sigaction(SIGALRM, &sa, NULL) < 0) {

perror("sigaction: sigalrm");

exit(1);

}

/\*

\* Ignore SIGPIPE, so that write()s to pipes

\* with no reader do not result in us being killed,

\* and write() returns EPIPE instead.

\*/

if (signal(SIGPIPE, SIG\_IGN) < 0) {

perror("signal: sigpipe");

exit(1);

}

}

static void do\_shell(char \*executable, int wfd, int rfd)

{

char arg1[10], arg2[10];

char \*newargv[] = { executable, NULL, NULL, NULL };

char \*newenviron[] = { NULL };

sprintf(arg1, "%05d", wfd);

sprintf(arg2, "%05d", rfd);

newargv[1] = arg1;

newargv[2] = arg2;

raise(SIGSTOP);

execve(executable, newargv, newenviron);

/\* execve() only returns on error \*/

perror("scheduler: child: execve");

exit(1);

}

/\* Create a new shell task.

\*

\* The shell gets special treatment:

\* two pipes are created for communication and passed

\* as command-line arguments to the executable.

\*/

static void sched\_create\_shell(char \*executable, int \*request\_fd, int \*return\_fd)

{

pid\_t p;

int pfds\_rq[2], pfds\_ret[2];

if (pipe(pfds\_rq) < 0 || pipe(pfds\_ret) < 0) {

perror("pipe");

exit(1);

}

p = fork();

if (p < 0) {

perror("scheduler: fork");

exit(1);

}

if (p == 0) {

/\* Child \*/

close(pfds\_rq[0]);

close(pfds\_ret[1]);

do\_shell(executable, pfds\_rq[1], pfds\_ret[0]);

assert(0);

}

/\* Parent \*/

close(pfds\_rq[1]);

close(pfds\_ret[0]);

\*request\_fd = pfds\_rq[0];

\*return\_fd = pfds\_ret[1];

root = ( struct procs \* )malloc(sizeof(struct procs) );

pointer = root;

start\_list=root;

previous=root;

pointer->next = NULL;

pointer->proc\_id = 0;

pointer->proc\_pid = p;

pointer->name = executable;

}

static void shell\_request\_loop(int request\_fd, int return\_fd)

{

int ret;

struct request\_struct rq;

/\*

\* Keep receiving requests from the shell.

\*/

for (;;)

{

if (read(request\_fd, &rq, sizeof(rq)) != sizeof(rq))

{

perror("scheduler: read from shell");

fprintf(stderr, "Scheduler: giving up on shell request processing.\n");

break;

}

signals\_disable();

ret = process\_request(&rq);

signals\_enable();

if (write(return\_fd, &ret, sizeof(ret)) != sizeof(ret))

{

perror("scheduler: write to shell");

fprintf(stderr, "Scheduler: giving up on shell request processing.\n");

break;

}

}

}

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

{

int i;

/\* Two file descriptors for communication with the shell \*/

static int request\_fd, return\_fd;

/\* Create the shell. \*/

sched\_create\_shell(SHELL\_EXECUTABLE\_NAME, &request\_fd, &return\_fd);

/\* TODO: add the shell to the scheduler's tasks \*/

/\*

\* For each of argv[1] to argv[argc - 1],

\* create a new child process, add it to the process list.

\*/

nproc = argc - 1; /\* number of proccesses goes here \*/

pid\_t p[nproc];

char \*newargv[] = { NULL, NULL, NULL, NULL };

char \*newenviron[] = { NULL };

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

{

/\* Creation of list\*/

root = ( struct procs \* )malloc(sizeof(struct procs) );

pointer->next = root;

pointer = root;

pointer->next = NULL;

p[i] = fork();

if( p[i] < 0 )

{

perror("fork");

exit(1);

}

else if ( p[i] == 0 )

{

/\* child \*/

raise(SIGSTOP);

newargv[0] = argv[i+1];

execve(argv[i+1], newargv, newenviron);

}

}

/\* Wait for all children to raise SIGSTOP before exec()ing. \*/

wait\_for\_ready\_children(nproc);

last\_p = pointer;

pointer = start\_list->next;

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

{

kill(p[i],SIGCONT);

pointer->proc\_pid = p[i];

pointer->proc\_id = i + 1;

pointer->name = argv[i+1];

pointer = pointer->next;

}

wait\_for\_ready\_children(nproc);

/\* Install SIGALRM and SIGCHLD handlers. \*/

install\_signal\_handlers();

pointer = start\_list;

previous = start\_list;

pointer = pointer->next;

kill(pointer->proc\_pid,SIGCONT);

alarm(SCHED\_TQ\_SEC);

if (nproc == 0) {

fprintf(stderr, "Scheduler: No tasks. Exiting...\n");

exit(1);

}

shell\_request\_loop(request\_fd, return\_fd);

/\* Now that the shell is gone, just loop forever

\* until we exit from inside a signal handler.

\*/

while (pause())

;

/\* Unreachable \*/

fprintf(stderr, "Internal error: Reached unreachable point\n");

return 1;

}

**Ερωτήσεις**

1. Όταν ο φλοιός υφίσταται χρονοδρομολόγηση, ως τρέχουσα διεργασία στη λίστα διεργασιών εμφανίζεται ο ίδιος ο φλοιός. Αυτό δεν θα μπορούσε να συμβαίνει διαφορετικά, γιατί η εντολή ‘p’ (εκτύπωση λίστας και τρέχουσας διεργασίας) εκτελείται από την διεργασία του φλοιού.
2. Είναι αναγκαίο να χρησιμοποιηθούν κλήσεις signals\_disable(), \_enable() γύρω από την συνάρτηση υλοποίησης αιτήσεων του φλοιού, έτσι ώστε κάθε αίτημα προς το φλοιό να ολοκληρώνεται με βεβαιότητα. Δηλαδή να μην μπορεί ένα άλλο σήμα (ούτε το SIGALARM) να διακόψει το αίτημα που εκτελείται, πράγμα που θα δημιουργούσε προβλήματα, αν μια αλλαγή στη λίστα δεν είχε ολοκληρωθεί.

**Άσκηση 1.3**

#include <errno.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <assert.h>

#include <stdbool.h>

#include <sys/wait.h>

#include <sys/types.h>

#include "proc-common.h"

#include "request.h"

/\* Compile-time parameters. \*/

#define SCHED\_TQ\_SEC 2 /\* time quantum \*/

#define TASK\_NAME\_SZ 60 /\* maximum size for a task's name \*/

#define SHELL\_EXECUTABLE\_NAME "shell" /\* executable for shell \*/

struct procs{

int proc\_id;

pid\_t proc\_pid;

char \*name;

int prio;

struct procs \*next;

};

struct procs \*root;

struct procs \*previous;

struct procs \*pointer;

struct procs \*start\_list;

struct procs \*last\_p;

struct procs \*print\_p;

struct procs \*temp\_p;

struct procs \*last\_high;

int nproc,identifier;

/\* Print a list of all tasks currently being scheduled. \*/

static void sched\_print\_tasks(void)

{

printf("Process that is currently running: %d\n", pointer->proc\_id);

print\_p = start\_list;

while(print\_p->next != NULL)

{

printf("Process id: %d\n", print\_p->proc\_id);

printf("Process pid: %d\n", print\_p->proc\_pid);

if(print\_p->proc\_id != 0)

printf("Process name: %s\n", print\_p->name);

else

printf("Process name: %s\n", print\_p->name);

printf("Process priority: %d\n", print\_p->prio);

print\_p = print\_p->next;

}

printf("Process id: %d\n", print\_p->proc\_id);

printf("Process pid: %d\n", print\_p->proc\_pid);

if(print\_p->proc\_id != 0)

printf("Process name: %s\n", print\_p->name);

else

printf("Process name: %s\n", print\_p->name);

printf("Process priority: %d\n", print\_p->prio);

print\_p = start\_list;

}

/\* Send SIGKILL to a task determined by the value of its

\* scheduler-specific id.

\*/

static int sched\_kill\_task\_by\_id(int id)

{

print\_p = start\_list;

bool flag3 = true;

while(print\_p->proc\_id != id)

{

print\_p = print\_p->next;

if(flag3 == true)

{

temp\_p = start\_list;

flag3 = false;

}

else

temp\_p = temp\_p->next;

}

kill(print\_p->proc\_pid, SIGTERM);

nproc--;

if(print\_p == previous)

{

temp\_p->next=print\_p->next;

previous = temp\_p;

}

else if(print\_p == pointer)

{

temp\_p=last\_p;

temp\_p->next=print\_p->next;

nproc++;

pointer = print\_p->next;

start\_list = pointer;

}

if(print\_p == last\_high)

{

temp\_p->next=print\_p->next;

if(temp\_p == start\_list)

last\_high = NULL;

else

last\_high = temp\_p;

}

free(print\_p);

print\_p = start\_list;

return 1;

}

/\* Create a new task \*/

static void sched\_create\_task(char \*executable)

{

pid\_t p;

char \*newargv[] = { executable, NULL, NULL, NULL };

char \*newenviron[] = { NULL };

p = fork();

if( p < 0 )

{

perror("fork");

exit(1);

}

else if ( p == 0 )

execve(executable, newargv, newenviron);

root = ( struct procs \* )malloc(sizeof(struct procs) );

last\_p->next = root;

previous = root;

last\_p = root;

last\_p->next = NULL;

root->name = malloc(sizeof(60));

last\_p->prio = 0;

last\_p->proc\_id = identifier + 1;

strcpy(last\_p->name,executable);

last\_p->proc\_pid = p;

nproc++;

identifier++;

}

static void high(int id)

{

print\_p = start\_list;

bool flag3 = true;

while(print\_p->proc\_id != id)

{

print\_p = print\_p->next;

if(flag3 == true)

{

temp\_p = start\_list;

flag3 = false;

}

else

temp\_p = temp\_p->next;

}

print\_p->prio = 1;

if(last\_high == NULL) //first high node

{

if(print\_p==last\_p)

{

last\_p=temp\_p;

}

temp\_p->next = print\_p->next;

last\_high = print\_p;

last\_high->next = start\_list->next;

start\_list->next = last\_high;

}

else if(print\_p == last\_p) //last low node becomes high

{

last\_p = temp\_p;

temp\_p->next = NULL;

print\_p->next = last\_high->next;

last\_high->next = print\_p;

last\_high = print\_p;

}

else

{

temp\_p->next = print\_p->next;

print\_p->next = last\_high->next;

last\_high->next = print\_p;

last\_high = print\_p;

}

print\_p = start\_list;

}

static void low(int id)

{

print\_p = start\_list;

bool flag3 = true;

while(print\_p->proc\_id != id)

{

print\_p = print\_p->next;

if(flag3 == true)

{

temp\_p = start\_list;

flag3 = false;

}

else

temp\_p = temp\_p->next;

}

print\_p->prio = 0;

if(print\_p == last\_high)

{

if(last\_high == start\_list->next)

{

last\_high = NULL;

}

else

{

last\_high = temp\_p;

last\_high->next = print\_p;

}

}

else

{

temp\_p->next = print\_p->next;

print\_p->next = last\_high->next;

last\_high->next = print\_p;

}

print\_p = start\_list;

}

/\* Process requests by the shell \*/

static int process\_request(struct request\_struct \*rq)

{

switch (rq->request\_no)

{

case REQ\_PRINT\_TASKS:

sched\_print\_tasks();

return 0;

case REQ\_KILL\_TASK:

return sched\_kill\_task\_by\_id(rq->task\_arg);

case REQ\_EXEC\_TASK:

sched\_create\_task(rq->exec\_task\_arg);

return 0;

case REQ\_HIGH\_TASK:

high(rq->task\_arg);

return 0;

case REQ\_LOW\_TASK:

low(rq->task\_arg);

return 0;

default:

return -ENOSYS;

}

}

/\*

\* SIGALRM handler

\*/

static void sigalrm\_handler(int signum)

{

if(signum != SIGALRM)

{

printf("Caught wrong signal");

exit(1);

}

else

{

kill(pointer->proc\_pid,SIGSTOP);

}

}

/\*

\* SIGCHLD handler

\*/

static void sigchld\_handler(int signum)

{

pid\_t p;

int status;

for(;;)

{

if(nproc == 0)

{

exit(1);

}

else

{

p = waitpid(-1, &status, WUNTRACED | WNOHANG);

if(p < 0)

{

perror("waitpid");

exit(1);

}

if(p == 0)

{

break;

}

explain\_wait\_status(p, status);

if(WIFEXITED(status) || WIFSIGNALED(status))

{

/\*A child has died\*/

if(pointer->next != NULL) //if we are not in the last node

{

if(pointer != start\_list) //if we are not in the first node

{

if(pointer == last\_high) //last node of high list

{

if(previous == start\_list)

{

last\_high=NULL;

}

else

last\_high = previous;

previous->next = pointer->next;

free(pointer);

pointer = start\_list;

previous = last\_p;

}

else

{

previous->next = pointer->next; //connect previous with next node

free(pointer);

pointer = previous->next;

} //pointer points to the next node

}

else //if we are in the first node

{

start\_list = pointer->next; //start\_list points to the second node

free(pointer);

pointer = start\_list; //pointer also points to the next node

}

}

else if( pointer->next == NULL ) //if we are in the last node

{

if(pointer == last\_high) //all nodes are high

{

if(previous == start\_list)

last\_high = NULL;

else

last\_high = previous;

previous->next = pointer->next;

free(pointer);

pointer = start\_list;

}

else

{

last\_p = previous;

previous->next = NULL; //previous points to the last node

free(pointer);

pointer = start\_list; //pointer points to the first node

}

}

nproc--;

kill(pointer->proc\_pid, SIGCONT);

alarm(SCHED\_TQ\_SEC);

}

if(WIFSTOPPED(status) )

{

/\*A child has stopped due to SIGSTOP/SIGTSTP, etc \*/

if(pointer->next != NULL && pointer != last\_high) //if we are not in the last node

{

pointer = pointer->next;

if(previous->next != NULL && previous != last\_high)

{

previous = previous->next;

}

else

{

previous = start\_list;

}

}

else

{

if(previous->next != NULL && previous != last\_high)

{

previous = previous->next;

}

else

{

previous = start\_list;

}

pointer = start\_list;

}

kill(pointer->proc\_pid, SIGCONT);

alarm(SCHED\_TQ\_SEC);

}

}

}

}

/\* Disable delivery of SIGALRM and SIGCHLD. \*/

static void signals\_disable(void)

{

sigset\_t sigset;

sigemptyset(&sigset);

sigaddset(&sigset, SIGALRM);

sigaddset(&sigset, SIGCHLD);

if (sigprocmask(SIG\_BLOCK, &sigset, NULL) < 0) {

perror("signals\_disable: sigprocmask");

exit(1);

}

}

/\* Enable delivery of SIGALRM and SIGCHLD. \*/

static void signals\_enable(void)

{

sigset\_t sigset;

sigemptyset(&sigset);

sigaddset(&sigset, SIGALRM);

sigaddset(&sigset, SIGCHLD);

if (sigprocmask(SIG\_UNBLOCK, &sigset, NULL) < 0) {

perror("signals\_enable: sigprocmask");

exit(1);

}

}

/\* Install two signal handlers.

\* One for SIGCHLD, one for SIGALRM.

\* Make sure both signals are masked when one of them is running.

\*/

static void install\_signal\_handlers(void)

{

sigset\_t sigset;

struct sigaction sa;

sa.sa\_handler = sigchld\_handler;

sa.sa\_flags = SA\_RESTART;

sigemptyset(&sigset);

sigaddset(&sigset, SIGCHLD);

sigaddset(&sigset, SIGALRM);

sa.sa\_mask = sigset;

if (sigaction(SIGCHLD, &sa, NULL) < 0) {

perror("sigaction: sigchld");

exit(1);

}

sa.sa\_handler = sigalrm\_handler;

if (sigaction(SIGALRM, &sa, NULL) < 0) {

perror("sigaction: sigalrm");

exit(1);

}

/\*

\* Ignore SIGPIPE, so that write()s to pipes

\* with no reader do not result in us being killed,

\* and write() returns EPIPE instead.

\*/

if (signal(SIGPIPE, SIG\_IGN) < 0) {

perror("signal: sigpipe");

exit(1);

}

}

static void do\_shell(char \*executable, int wfd, int rfd)

{

char arg1[10], arg2[10];

char \*newargv[] = { executable, NULL, NULL, NULL };

char \*newenviron[] = { NULL };

sprintf(arg1, "%05d", wfd);

sprintf(arg2, "%05d", rfd);

newargv[1] = arg1;

newargv[2] = arg2;

raise(SIGSTOP);

execve(executable, newargv, newenviron);

/\* execve() only returns on error \*/

perror("scheduler: child: execve");

exit(1);

}

/\* Create a new shell task.

\*

\* The shell gets special treatment:

\* two pipes are created for communication and passed

\* as command-line arguments to the executable.

\*/

static void sched\_create\_shell(char \*executable, int \*request\_fd, int \*return\_fd)

{

pid\_t p;

int pfds\_rq[2], pfds\_ret[2];

if (pipe(pfds\_rq) < 0 || pipe(pfds\_ret) < 0) {

perror("pipe");

exit(1);

}

p = fork();

if (p < 0) {

perror("scheduler: fork");

exit(1);

}

if (p == 0) {

/\* Child \*/

close(pfds\_rq[0]);

close(pfds\_ret[1]);

do\_shell(executable, pfds\_rq[1], pfds\_ret[0]);

assert(0);

}

/\* Parent \*/

close(pfds\_rq[1]);

close(pfds\_ret[0]);

\*request\_fd = pfds\_rq[0];

\*return\_fd = pfds\_ret[1];

root = ( struct procs \* )malloc(sizeof(struct procs) );

pointer = root;

start\_list = root;

previous = root;

pointer->next = NULL;

pointer->proc\_id = 0;

pointer->proc\_pid = p;

pointer->name = executable;

}

static void shell\_request\_loop(int request\_fd, int return\_fd)

{

int ret;

struct request\_struct rq;

/\*

\* Keep receiving requests from the shell.

\*/

for (;;)

{

if (read(request\_fd, &rq, sizeof(rq)) != sizeof(rq))

{

perror("scheduler: read from shell");

fprintf(stderr, "Scheduler: giving up on shell request processing.\n");

break;

}

signals\_disable();

ret = process\_request(&rq);

signals\_enable();

if (write(return\_fd, &ret, sizeof(ret)) != sizeof(ret))

{

perror("scheduler: write to shell");

fprintf(stderr, "Scheduler: giving up on shell request processing.\n");

break;

}

}

}

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

{

int i;

/\* Two file descriptors for communication with the shell \*/

static int request\_fd, return\_fd;

/\* Create the shell. \*/

sched\_create\_shell(SHELL\_EXECUTABLE\_NAME, &request\_fd, &return\_fd);

/\* TODO: add the shell to the scheduler's tasks \*/

/\*

\* For each of argv[1] to argv[argc - 1],

\* create a new child process, add it to the process list.

\*/

nproc = argc - 1; /\* number of proccesses goes here \*/

identifier=nproc;

pid\_t p[nproc];

char \*newargv[] = {NULL, NULL, NULL, NULL};

char \*newenviron[] = { NULL };

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

{

/\* Creation of list\*/

root = ( struct procs \* )malloc(sizeof(struct procs) );

pointer->next = root;

pointer = root;

pointer->next = NULL;

root->name = malloc(sizeof(60));

p[i] = fork();

if( p[i] < 0 )

{

perror("fork");

exit(1);

}

else if ( p[i] == 0 )

{

/\* child \*/

raise(SIGSTOP);

newargv[0] = argv[i+1];

execve(argv[i+1], newargv, newenviron);

}

}

/\* Wait for all children to raise SIGSTOP before exec()ing. \*/

wait\_for\_ready\_children(nproc);

last\_p = pointer;

last\_high = NULL;

pointer = start\_list->next;

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

{

kill(p[i],SIGCONT);

pointer->proc\_pid = p[i];

pointer->proc\_id = i + 1;

strcpy(pointer->name, argv[i+1]);

pointer->prio = 0;

pointer = pointer->next;

}

wait\_for\_ready\_children(nproc);

/\* Install SIGALRM and SIGCHLD handlers. \*/

install\_signal\_handlers();

pointer = start\_list->next;

previous = start\_list;

kill(pointer->proc\_pid,SIGCONT);

alarm(SCHED\_TQ\_SEC);

if (nproc == 0) {

fprintf(stderr, "Scheduler: No tasks. Exiting...\n");

exit(1);

}

shell\_request\_loop(request\_fd, return\_fd);

/\* Now that the shell is gone, just loop forever

\* until we exit from inside a signal handler.

\*/

while (pause())

;

/\* Unreachable \*/

fprintf(stderr, "Internal error: Reached unreachable point\n");

return 1;

}

**Ερωτήσεις**

1. Ένα σενάριο λιμοκτονίας είναι μία διεργασία που βρίσκεται σε high priority να μην τερματίζει ποτέ, με αποτέλεσμα όλες οι διεργασίες που βρίσκονται σε low priority να μην εκτελεστούν καθόλου.